Using Domain Knowledge to Enhance Process Mining Results

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 algorithms achieve this by solely using an event log, without allowing the domain expert to influence the discovery in any way. However, the user may have certain domain expertise which should be exploited to create better process models. In this paper, we address this issue of incorporating domain knowledge to improve the discovered process model. First, we present a verification algorithm to verify the presence of certain constraints in a process model. Then, we present three modification algorithms to modify the process 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 [23]. 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 [2]. Whereas enhancement techniques use event data and process models to repair or enrich the process model. Process models can be represented by multiple modeling notations, for example BPMN, Petri nets, process trees, etc.

Most often, the focus of process discovery techniques is on automatically extracting information to discover models solely based on the event logs. Hence, in order to gain significant outcomes, the event log should ideally contain all the necessary information required by the algorithm. However, in many real world

scenarios, the data contained in the event logs may be incomplete and/or noisy. This could in turn result in incorrect interpretations of the data. One of the ways to circumvent the problems with insufficient data could be to allow the user to input certain expert knowledge. The domain knowledge could thus be used to overcome the shortcoming of the data; thereby improving the results. In this paper, we address this challenge of incorporating domain knowledge in traditional process discovery.

Domain knowledge can be introduced in the discovery process at multiple stages as shown in Figure 1. In this paper, we focus on *post-processing* an already discovered process model to incorporate the user specified domain knowledge. The primary reason of introducing domain knowledge at the post-processing stage is to keep the overall method generic and scalable. Our approach can be coupled with *any* discovery algorithm which generates models which can be represented as process trees. Alternatively, we can refine (enhance) a pre-existing model using our technique. We focus on a class of models represented as process

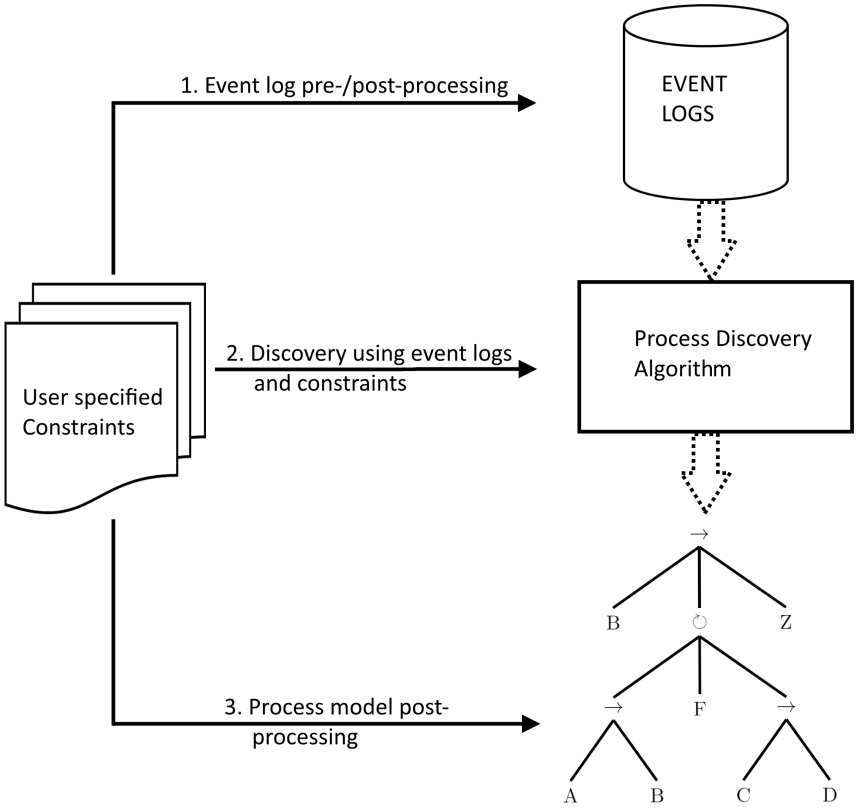


Fig. 1: Places where domain knowledge can be incorporated in the process discovery approach. In this paper we focus on post-processing the model based on domain knowledge (approach 3).

trees because the state of the art discovery algorithms such as the Inductive Miner [16] and the Evolutionary Tree Miner [5] discover block-structured process models represented by *process trees*. Furthermore, process trees 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.

We use *Declare* templates [19] as a way to input the domain knowledge. *Declare* templates belong to the class of declarative languages, which are used to construct constraint based declarative process models. We abstract these templates as a way to specify domain knowledge effectively in terms of constraints. Declare provides a handy and effective way of modeling protocols and rules. Such constrains are especially efficient in many real life domains, such as healthcare wherein the the clinical specialist are usually aware of certain protocols which should hold in the process. Using a subset of declare templates the user can specify a set of constraints which must hold in the process.

In order to incorporate domain knowledge in the post-processing stage, we primarily focus on addressing two main challenges. Firstly, we introduce a verification algorithm, to determine whether a set of user specified constraints is satisfied in the model. We then introduce and evaluate three modification techniques in order to generate variants of the input process model based on the constraints and event log. In [10], we introduced the verification and a brute force modification approach. We extend that work and introduce two additional modification approaches: genetic and constraint specific modification. All three modification algorithms output a number of process tree variants. Each of these variants may satisfy the user specified constraint to a different degree. The user can choose the most appropriate model depending on the constraints satisfied, along with the values of four quality dimensions (replay fitness, precision, simplicity and generalization).

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 verification and modification algorithms. In Section 6 we evaluate our approach qualitatively as well as quantitatively 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 [1, 2, 8] 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 a process model, but do not provide any way of incorporating domain knowledge to repair/improve the process model. [20] provides a backward compliance checking technique using alignments wherein the user provides compliance rules as Petri-nets. The focus is on using Petri-net rules for diag-

nostic analysis on the event log, rather than discovering a new process model with compliance rules. The conformance based repair technique suggested by [11] 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.

[12] and [22] propose approaches to verify the presence/absence of constraints in the process model. However, these approaches do not provide a way to modify the process models w.r.t. the constraints. In [18], the authors provide a way to mine declarative rules and models in terms of LTL formulas based on event logs. Similarly, [7] uses declare taxonomy of constraints for defining artful processes expressed through regular expressions. [15] uses event logs to discover processes models using Inductive Logic Programming represented in terms of a sub-set of SCIFF ([3]) language, used to classify a trace as compliant or non compliant. In [6], the authors extend this work to express the discovered model in terms of Declare model. However, these approaches focus on discovering rules/constraints from event log without allowing the users to introduce domain knowledge during rule discovery.

In [21], 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. [24] proposes a discovery algorithm using Integer Linear Programming based on the theory of regions, which can be extended with a limited set of user specified constraints during process discovery. In [13], the authors introduce a process discovery technique presented as a multi-relational classification problem on event logs. Their approach is supplemented by Artificially Generated Negative Events(AGNEs), with a possibility to include prior knowledge (e.g. causal dependencies, parallelism) during discovery. The authors of [14] incorporate both positive and negative constraints during process discovery to discover C-net models. Compared to [13], [14], and [24], 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.

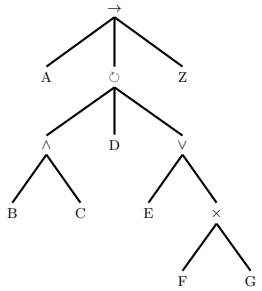
3 Preliminaries

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

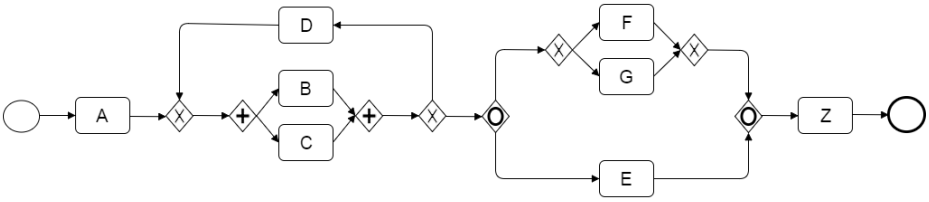
3.1 Process Trees

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 2 shows an example process tree and its equivalent BPMN model. A process tree is traversed from left to right and top to bottom.

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 2a, activities A and Z are always the first and last activities respectively. For the $\circlearrowleft$ operator the left most node is the 'do' part of the loop and is executed at least once. In Figure 2a, activity D is the optional 're-do' part of $\circlearrowleft$, 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 $\times$ operator, only one of either F or G is chosen. For the $\vee$ operator both $\times$ and activity E can occur in any order, or only one of either two can occur.



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



(b) BPMN equivalent of the process tree from Figure 2a

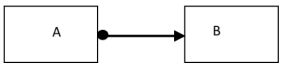
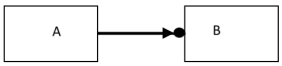
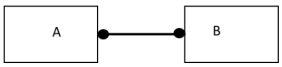
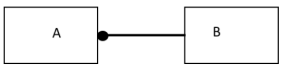
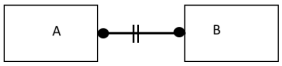
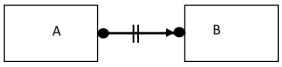
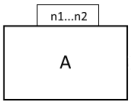
Fig. 2: Example process tree and its BPMN equivalent.

3.2 Declare Templates

A declarative model is defined by using constraints specified by a set of templates [19]. 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 [18, 19]. Binary templates provide ways to specify dependency

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

(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.

4 Verification

In this section, we present a novel verification approach that takes a process tree and a set of constraints as input, and returns the set of constraints satisfied by the process tree as output. We recall from subsection 3.1 that the order of children for the operators $\vee$, $\times$,and $\wedge$ is not fixed and the child nodes can be executed

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             foreach sub-tree do
 6                 verify common parent
 7                 verify position of activties
 8                 if common parent or position verification fails then
 9                     set constraint verification unsuccessful
10                 else if relations constraint & occurs(ST_A,B) is (always) then
11                     set constraint verification successful
12                 else if negative relations constraint & occurs(ST_A,B) is (never)
                   then
13                     set constraint verification successful
14                 else
15                     set constraint verification unsuccessful
16         else
17             consider full tree
18             check range from occurs(PT,A) to occurs_multiple_times(PT,A)
19     return set of constraints marked as - verified or unverified
```

in any order. However, for the remaining nodes, ($\rightarrow$, $\circlearrowleft$) the order of navigation is fixed from left to right. We split the verification procedure into three parts. First, we check if the positioning of the activities in the process tree is correct according to the constraint. For example, for a constraint $response(A, B)$, we check if all the B's are on the *right* side of A. Next, we perform an additional check to verify the common parent, addressed in Subsection 4.2. In the case of $response(A, B)$, even if activity B is on the *right* side of activity A, the order isn't fixed if the common parent between A and B is not $\rightarrow$ or $\circlearrowleft$ and hence B is not guaranteed to occur after A. Finally, we calculate the occurrence possibilities of B w.r.t. the common parent. In Algorithm 1, we show the main sequence of steps used by the verification approach. In the following sub-sections, we detail the algorithm.

4.1 Sub-tree Computation & Position Verification

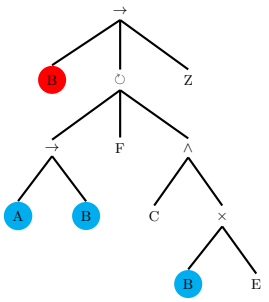
In this sub-section we explain the sub-tree computation and position verification with respect to each of the binary constraints. Sub-trees are sub-blocks containing the first common ancestor between the two activities of the binary (negative) relation constraints. The same activity can be present at multiple locations in a

Table 2: Primary activities corresponding to binary constraints

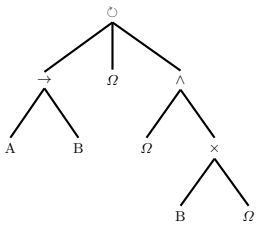
constraint	primary activity
$response(A, B)$	A
$precedence(A, B)$	B
$coexistence(A, B)$	A,B
$responded - existence(A, B)$	A
$not - succession(A, B)$	A
$not - coexistence(A, B)$	A,B

process tree which could result in multiple sub-trees for a single constraint. The total number of sub-trees is equal to the number of occurrences of the 'primary' activity from the constraint in the process tree. The primary activity is the activity w.r.t. which the constraint is specified. For e.g., for constraint *response(A,B)*, the primary activity is 'A'. For constraint such as *coexistence(A,B)*, where both the activities A and B are primary activities, the total number of sub-trees computed is $n + m$, where n and m are the number of occurrences of activities A and B in the process tree respectively. Table 2 provides the overview of primary activities corresponding to each constraint.

For sub-tree calculation, we make use of the fact that a process tree is generally navigated from *left to right* for $\rightarrow$ and $\circlearrowleft$, and that the ordering doesn't matter for other operators. This provides a good starting point to compare the ordering of activities in a process tree with respect to the declare constraints. Consider the constraint $response(A, B)$ that should be verified for the process tree from Figure 3a. As described in Table 1, a response constraint states 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 discussed earlier, all the B's that could occur after A have to be on the *right* side of A. Figure 3b shows the sub-tree for constraint $response(A, B)$. Since there is only



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



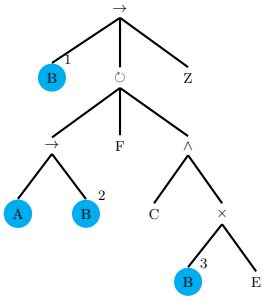
(b) Sub-tree for $response(A, B)$ with all the irrelevant activities marked as Ω

Fig. 3: Sub-tree computation for the constraint $response(A, B)$

one occurrence of activity A in the process tree, there is only one sub-tree. The first occurrence of B from the 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 4. The sub-trees for B_2 and B_3 are shown in Figure 4c and Figure 4d 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 4b, 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 5 shows the sub-trees for constraints $responded - existence(A, B)$ and $coexistence(A, B)$. The sub-tree in Figure 5d is calculated with respect to activity B and is only valid for the constraint $coexistence(A, B)$. Negative relations constraints are more restrictive and the sub-trees can be calculated and derived in a similar way to their respective relation constraints counterpart. For example, for the constraint $not - coexistence(A, B)$, the sub-trees can be computed similar to $coexistence(A, B)$. However, unlike relation constraints, for negative relation 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 if the common parent is valid as discussed in Subsection 4.2.



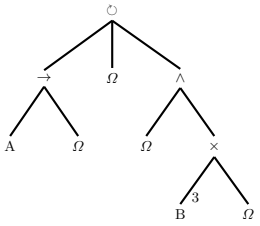
(a) $Precedence(A, B)$ - Active nodes for this constraint are highlighted in cyan

null

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

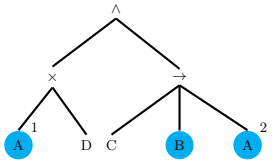


(c) $Precedence(A, B)$ - Sub-tree with respect to activity B_2

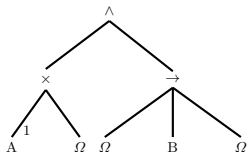


(d) $Precedence(A, B)$ - Sub-tree with respect to activity B_3

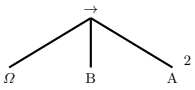
Fig. 4: Sub-trees computation for the constraint $precedence(A, B)$. As Figure 4b results in a **null** sub-tree, the constraint verification for the constraint $precedence(A, B)$ fails w.r.t. the entire process tree.



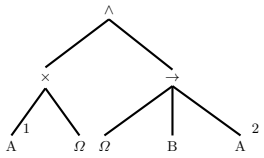
(a) *Responded* − *existence*(A, B) and *coexistence*(A, B) - Active nodes for these constraints are highlighted in cyan



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

4.2 Parent Verification

If sub-tree computation 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 from Figure 3a, then one of the sub-trees computed is shown in Figure 6. As the common parent for this sub-tree is the choice operator ×, both E and B will never occur together. Hence the common parent verification for this particular sub-tree fails for constraint *coexistence*(E, B). Parent verification is not required for unary constraints, as there is only one activity. Table 3 summarizes the valid common parents for all the binary constraints from Table 1.



Fig. 6: Sub-tree violating constraint *coexistence*(E, B)

4.3 Activity Occurrence Verification

For binary constraints the next step 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*.

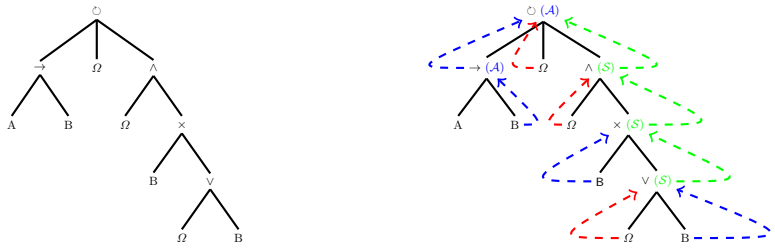
Figure 7b shows the occurrence of activity B, for the sub-tree from Figure 7a which is computed with respect to activity A. For choice operators such as × and

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

constraint	valid common parent operator
$response(A, B)$	$\rightarrow$, $\circlearrowleft^1$
$precedence(A, B)$	$\rightarrow$, $\circlearrowleft^1$
$coexistence(A, B)$	$\rightarrow$, $\wedge$, $\circlearrowleft^1$
$responded-existence(A, B)$	$\rightarrow$, $\wedge$, $\circlearrowleft^1$
$not-succession(A, B)$	$\times$
$not-coexistence(A, B)$	$\times$
	$\circlearrowleft^1$ is valid only if node B (or A) is a child of the left (do) or right (exit loop) part and *not* of middle(re-do) part

$\vee$, 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 child nodes 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. Starting with the deepest occurrence of B in sub-tree, we annotate the 'occurrence of B' w.r.t. each node recursively until the root of the sub-tree is reached as shown in Figure 7b. 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.

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



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

Table 4: 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

occurrence of activity A is set to *yes*. Otherwise, the multiple occurrence part of 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 4.

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

5 Modification

Following the description of the verification algorithm, we now discuss three modification approaches in order to incorporate the user specified domain knowledge in the process tree. Figure 8 gives a general overview of the overall approach, independent of the modification algorithm used. The three modification algorithms proposed are : brute force modification, genetic modification, and constraint specific modification. The brute force modification approach used only a process tree during modification. The constraint specific modification algorithm uses the user specified constraints along with the initial process tree during modification. The genetic modification approach requires the initial process tree, user specified constraints as well as the event log as during modification. It should be noted that in order to compute the final Pareto front, all the three techniques use the event log and the user specified constraints.

All modification algorithms produce a collection of candidate process trees as output. Each candidate process tree is evaluated against the four quality dimensions of process mining (replay fitness, precision, generalization and simplicity) [5, 23] and the number of user specified constraints verified by the tree. This results in five quality dimensions. In order to evaluate the process trees based on these dimensions we use a Pareto front [5]. The general idea of a Pareto front

is that all models are mutually non-dominating: A model is dominating with respect to another model, if for all measurement dimensions it is at least equal or better and for one strictly better. Using the five dimensions, a Pareto front is presented to the user which contains the set of dominating process trees.

5.1 Brute Force Modification

The brute force modification algorithm takes the discovered process tree as input and generates a list of candidate trees using a brute force approach. This is accomplished as shown in Algorithm 2, elaborated in 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 new variants of process trees. After each edit operation, the number of edits for the process tree is incremented. It should be noted that, in case of leaf nodes, *all* the activities are used exhaustively for *each* type of edit operation. For example, in Figure 9c, activity A is added as a left-most child of $\times$, resulting in a new process tree. Similarly, activity B would also

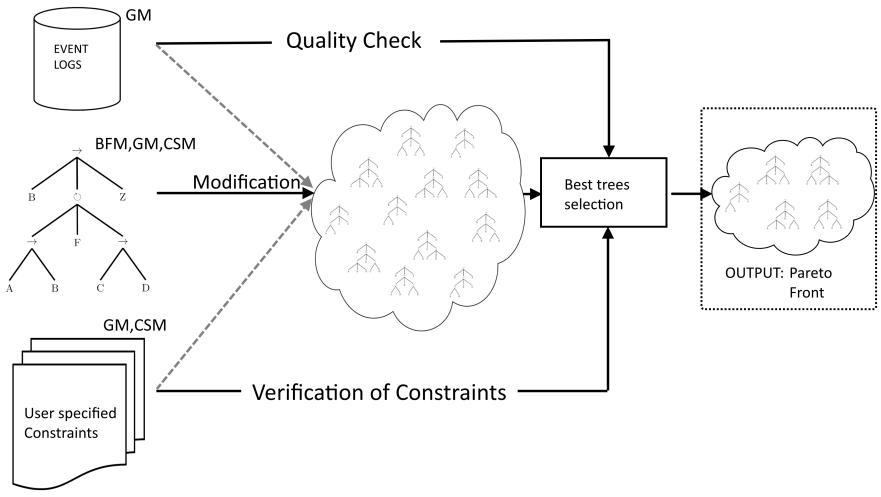


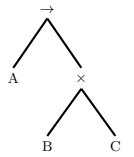
Fig. 8: The general overview combining traditional process discovery with domain knowledge specified using constraints. The figure also shows the inputs required by each modification approach - brute force modification(BFM), genetic modification(GM), and constraint specific modification(CSM). Each of the modification algorithms creates variants of process trees, after which the best process trees are selected using a Pareto front based on the four quality dimensions of process mining and number of constraints satisfied.

Algorithm 2: Brute force modification of a process tree

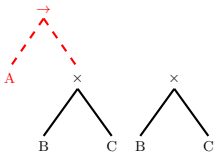
Input: process tree (PT), max nbr of edits(N_{max})
Output: candidate process trees collection(C)

```
1  begin
2      Create candidate process trees collection C
3      C= {pt} ∪ ModifyProcessTree(PT, 0, N_max)
4      return C;
5  Function ModifyProcessTree(pt,n,N_max)
6      Initiate local process trees collection C'
7      if n < N_max then
8          foreach node ∈ process tree pt do
9              foreach edit operator do
10                 create p̃t from pt by using the edit operator for the chosen node
11                 reduce tree p̃t if possible
12                 C' = C' ∪ {p̃t}
13                 C' = C' ∪ ModifyProcessTree(p̃t, n + 1, N_max)
14     return C'
```

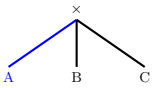
be added as a left-most child, resulting in another process tree and same goes for activity C. Moreover, each activity is also added to every possible location within the tree. For example in Figure 9c, activity A could be added



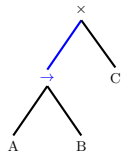
(a) Input Process tree. *(number of edits: 0)*



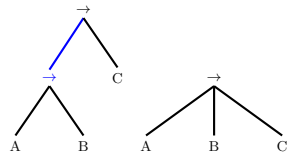
(b) Remove node: Removed activity A. Since → has only one child, it can be reduced. *(number of edits: 1)*



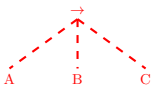
(c) Add node: Added activity A as a child of × *(number of edits: 2)*



(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. 9: Example modification operations on process tree.

in between B and C resulting in a new process tree, as well as activity A would be added after C (*right of C*) resulting in a new process tree. Similarly, in case of operator nodes, *all* the other operator nodes are used exhaustively in each edit operation.

3. After every edit operation, the newly created tree is reduced recursively until no reductions are possible. The reduction primarily consists of the following two actions:
   - If an operator node contains only one child, then remove the operator node and replace it with its child node. (e.g. Figure 9b)
   - If a parent and a child are of the same operator type in $\rightarrow, \wedge, \times$ or $\vee$; then the parent and child nodes can be collapsed into one operator. (e.g. Figure 9e)
4. Each newly created variant of the process tree is further edited by iteratively calling all the edit operations exhaustively (in all possible orders) on each node using a "brute force" approach.
5. This process of creating process tree variants is repeated until all the nodes of all the process trees are processed and/or the threshold for the *number of edit* operations w.r.t. the process tree is reached.
6. Every variant of the process tree is added to the pool of candidate process trees. The candidate process trees might contain duplicates. The brute for approach could be computationally expensive, hence adding further processing to check for duplicates in a pairwise manner for all the candidate process trees would further degrade the performance. Therefore we do not remove the duplicates. However, the Pareto front presented to the user would take care of these duplicates from the candidate process trees.

Figure 9 shows different edit operations used by the modification algorithm. The *Modify node* operation modifies every node in the process tree and can be classified into *Modify activity* and *Modify operator* depending on the type of node selected. Similarly, *Add node* adds either an activity node or an operator node (Figure 9c and Figure 9d). An operator can be added below the parent node (Figure 9d) and above the parent node (not shown in Figure 9) by exhaustively combining child nodes. Each edit operation 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.

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. Although a high threshold would explore more possibilities, it would result in a large number of variants of the process tree, and hence it would also be very computationally intensive and inefficient. Therefore, the threshold for number of edit operations should be chosen by the user depending on the original (starting) process tree, and the number of unverified constraints

in the original process tree. That is, if the originally discovered process tree is completely inappropriate according to the domain expert, and the number of unverified constraints is high, then the threshold should be set high. Contrary to this, if the domain expert agrees with major elements of originally discovered process tree, and the number of unverified constraints is low, then the threshold should be set low.

5.2 Genetic Modification

The brute force approach from Subsection 5.1 takes as input the process tree, and creates variants of this process tree in a brute force way, without considering the constraints or data. This approach takes into account the complete search space within the threshold of the number of edits. However, this could be unnecessarily time-consuming, as variants of process trees might be created iteratively which do not satisfy any constraints at all, or variants of process trees which do not describe the event log very well, or both. In order to overcome this, we present a second modification algorithm using a genetic approach. The creation of process trees is guided by the event log using the four quality dimensions (replay fitness, precision, generalization and simplicity) along with the number of constraints verified according to the verification algorithm.

We extend the genetic approach for process trees discussed in [4] to include an additional fitness evaluator "number of constraints specified", along with the four standard quality dimensions. A number of factors such as crossover and mutation probabilities, number of random trees in each generation etc. determine the variation of process trees in each generation. The tree modification process is guided to balance between the standard quality dimensions (determined by the event log) and the number of constraints verified. The modification of the trees is stopped when the stopping criteria of the genetic algorithm are met. The stopping criteria could be the maximum number of generations, maximum time taken, minimum number of constraints satisfied etc. The end result is a Pareto front containing the best process trees balanced on the five dimensions.

5.3 Constraint Specific Modification

The modification approaches discussed in Subsection 5.1 and Subsection 5.2 first modify the process trees (either in a brute force way or genetically), and then evaluate them to check the compliance of user specified constraints on the modified trees. An alternative to this approach would be to change the process trees according to the type of constraint, rather than changing the trees first and then verifying. In order to achieve this, we introduce a constraint specific modification approach which iteratively changes the process trees for each constraint specified by the user. The resulting process trees are subject to modification with respect to the next constraint considering all possible permutations. The current implementation of constraint specific modification requires the input process tree to have non-duplicate labels. In future we aim to overcome this issue of duplication

by taking in account all the combinations of each activity occurrence in the process tree, depending on the type of constraint. However due to the complexity and search space, we limit the current implementation to non-duplicate labels. The steps followed for constraint specific modification are mentioned below:

1. As a first step, all the possible permutations of the constraints given by the user are computed. For example, if there are two constraints input by the user - *response(A,B), precedence(C,D)* then the set of possible permutations of these constraints is - (<*response(A,B)*, *precedence(C,D)*>; <*precedence(C,D)*, *response(A,B)*>)
2. For each permutation, the process tree is modified with respect to each constraint in a sequential order as shown in Figure 10. Hence, new variants of process trees, are created and added to the output process trees pool. Each new variant is again subject to further modification, based on the next constraint in the set. The modification of process tree(s) based on the type of constraint is performed using following step(s):
   (a) For unary constraints, the activity from the constraint is mapped to the activity in the process tree. As discussed previously, we assume that every activity in a process tree is present at most once. Next, according to Table 4 if *occurs(PT,A)* of the activity is *sometimes*, then we introduce a

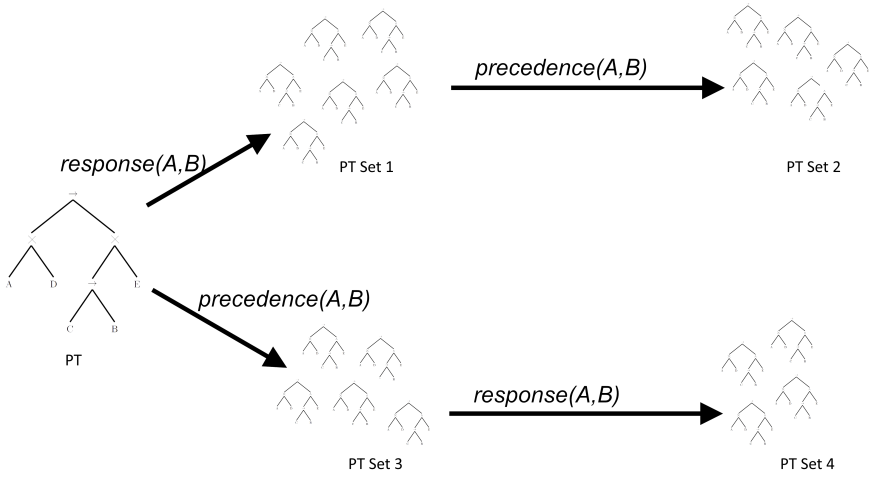


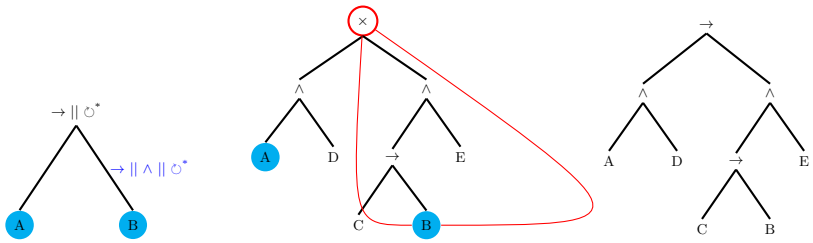
Fig. 10: High level representation of constraint specific modification approach. Starting with initial process tree, multiple variants are created w.r.t. each constraint. Every edited process tree is subject to the next constraint, until all the constraints are considered. This process is repeated for all the possible combinations of constraints. All the process trees in each PT Set are added to the final output of candidate trees for evaluation in the Pareto front.

Table 5: Valid intermediate operators for each of the declare constraints

constraint	valid common parent operator
$response(A, B)$	$\rightarrow$, $\wedge$, $\circlearrowleft^1$
$precedence(A, B)$	$\rightarrow$, $\wedge$, $\circlearrowleft^1$
$responded - existence(A, B)$	$\rightarrow$, $\wedge$, $\circlearrowleft^1$
$not - succession(A, B)$	$\times$, $\rightarrow$, $\wedge$, $\circlearrowleft$, $\vee$ *(all operators)*
$not - coexistence(A, B)$	$\times$, $\rightarrow$, $\wedge$, $\circlearrowleft$, $\vee$ *(all operators)*
	$\circlearrowleft^1$ is valid only if node B (or A) is a child of the left (do) or right (exit loop) part and *not* of middle(re-do) part

choice operator as one of the ancestor of activity A (if it doesn't already exist at any ancestors). Each possible ancestor is changed, resulting in creation of new process trees. Each newly created process tree is added to the pool of candidate process trees. Similarly, if *occurs(PT,A)* of the activity is *always*, then we replace every choice operator (one by one) with allowable operators ($\rightarrow$,$\wedge$,$\circlearrowleft$). Each resulting process tree is added to the pool of candidate process trees. Similar steps are repeated depending on the values of *occurs_multiple_times(PT,A)* and the presence/absence of loop operator in the ancestors of activity A. In case of binary constraints, we follow steps (b) to (e).

(b) For binary constraints, two activities from the constraint are used to create a process tree as shown in Figure 11a. The root shows the allowable operators as common ancestor. The operators on the edge show the allowable intermediate operators between the root and node B.



(a) Process tree showing allowable operators for constraint *response(A,B)*.

(b) Process tree highlighted with red as the common parent operator. The red area shows the intermediate operators between common parent and activity B.

(c) Modified process tree, satisfying the constraint *response(A,B)*, formed by combining process tree from Figure 11a and Figure 11b (by replacing $\times$ with $\rightarrow$).

Fig. 11: Constraint specific modification for *response(A,B) - depicted as a process tree in Figure 11a* on the process tree from Figure 11b

(c) The common ancestor between these activities is checked. If the common ancestor between these activities is valid (as described in Table 3), then proceed to next the step. If the common ancestor of is not valid, then it is changed to an allowable valid operator from Table 3. This is done for every allowable valid operator and results in a set of process trees. Each process tree created is subject to the next steps.

(d) The next step is to check the intermediate operators between common parent and the secondary activity from the constraint in the process tree as shown in Figure 11b. Constraint such as *co-existence(A,B)* is decomposed into two constraints *responded-existence(A,B)* and *responded-existence(B,A)*, which essentially add up to *co-existence(A,B)*. The allowable valid operators between common ancestor and secondary activity are mentioned in Table 5.

(e) In case the intermediate operator is not valid, it is replaced by each valid intermediate operator from Table 5. Every replacement of the operator results in a new process tree, which is added to the list of candidate process trees. Figure 11 shows the constraint specific modification process for the constraint *response(A,B)*. Figure 11c shows *one* of the outcomes of the constraint specification modification. It should be noted that some constraints share similar properties.

3. Finally, the best variants are chosen using a Pareto front made of the number of constraints satisfied and the four standard quality dimensions of process mining. It should be noted that some of the constraints might share similar properties or might be related. In such cases, for certain combinations, modification of a tree to satisfy a certain constraint might inherently also lead to the satisfaction of another constraint. In such scenarios, there is no further modification required, if there are no unsatisfied constraints remaining for the current combination set. For example, the valid common operators as shown in Table 3 for $response(A, B)$ and $precedence(A, B)$ are same. Therefore if a process tree contains a single occurrence of activities A and B, then the modification to change the common parent for the constraint $response(A, B)$, would automatically contain the modifications for the constraint $precedence(A, B)$ too. Hence step (c) could be skipped when the modifications for $prcedence(A, B)$ are being performed.

6 Evaluation

The outcomes of the three modification approaches can be evaluated primarily in two ways. One way is to qualitatively evaluate the differences in each modification approach in terms of configuration, performance and output. The second approach is to quantitatively evaluate the outcomes of the three modification approaches, and to specifically determine the added benefit that domain knowledge brings in improving the process models. We first compare and discuss the peculiarities of the three modification approaches qualitatively based on different parameters. Next, we evaluate the outcomes of different modification approaches quantitatively using a synthetic and a real life event log.

Table 6: Input and input parameters affecting the discovery outcome for each modification approach.

Constraint specific	Brute force	Evolutionary tree miner
event log	event log	event log
constraints	constraints	constraints
process tree	process tree	process tree
	threshold for # of edits	population size
		elite count
		max generations
		max duration
		cross over probability
		mutation probability

6.1 Comparison of Approaches

Multiple parameters determine the outcome of each technique. Every algorithm has various pro's and con's associated with it. That is, one technique might excel in a certain scenario where another technique might fail and vice versa. Therefore it is important to first carefully evaluate each approach qualitatively to analyze and compare the properties associated with each. In this sub section we focus on the qualitative comparison of the three modification algorithms introduced in this paper.

Configuration This section addresses the inputs and the configurable input parameters required by the modification techniques, which have a direct or indirect impact on the outcome of the algorithm. As discussed previously, all the three techniques require an event log, user specified constraints and a process tree as input. The constraint-specific modification technique does not require any additional inputs. The brute force modification technique requires the user to set the 'threshold for # of edits'. This threshold sets the maximum space explored by the algorithm, and hence has a big impact on the number of trees discovered. The genetic approach requires additional parameters which are standard for a genetic algorithm such as - maximum population, elite size, mutation probabilities etc. It should also be noted that the genetic algorithm is probabilistic whereas brute force and constraint specific modification techniques are deterministic. Table 6 gives an overview of the inputs required by the three techniques.

Event Log usage The way in which the information from event logs is used by the three approaches varies. Table 7 shows the usage of event logs in each approach. The brute force and the constraint specific modification techniques first create a set of candidate process trees without using any information from the event log. The event log is only used at a later stage to compute the quality scores with respect to each process tree variant, which is in turn used to construct a Parteo front. The genetic algorithm on the other hand uses the event logs along with the number of constraints verified as the guiding rule. Each process tree

Table 7: Usage of event logs of different algorithms.

Event log used during	Constraint Specific	Brute Force	Genetic
final evaluation	✓	✓	✓
intermediate evaluation	×	×	✓

variant created is subject to log alignments. The majority of the trees which make it to the next generation score well in some (or all) of the quality dimensions.

Time Performance The performance time of each algorithm varies depending on the size of input process tree, size of the event logs and most importantly, the individual algorithm settings. All the three modification approaches are subject to constraint verification and event log alignments as final evaluation to calculate the quality of candidate process trees. It should be noted that in the case of genetic modification there are also intermediate event log alignments with respect to every intermediate process tree. The genetic algorithm additionally depends on the setting of various parameters. For example, if the the parameters such as population size and maximum generations are kept high, the genetic algorithm may take a long time to complete. The time complexity in the case of brute force modification technique depends on the 'threshold for maximum number of edits' and the number of nodes in the initial tree. However, the time complexity for constraint specific modification approach is determined by the number of constraints, the type of constraints and the depth of the initial tree. That is, if the number of constraints are high, the constraints are unrelated and the size of the initial tree is big then the constraint specific algorithm can take a long time to complete. In a real life setting, the constraint specific modification is usually faster than the genetic approach, which in turn is faster than brute force modification. This can be explained by the fact that constraint specific modification is primarily dependent on the number of constraints, hence is fastest. The genetic approach discards the uninteresting candidates in each generation, thereby converging faster than the brute force approach, which explores all the variants within the given threshold. Of course, as mentioned previously the individual parameters have a much higher impact on the performance time of each of the approaches.

Next, we focus on discussing the quantitative evaluation of the candidate process trees from the Pareto front. One method to evaluate the quality of outcome from each modification approach could be to present the domain expert with a list of candidate process trees (or process models) to choose from. The domain expert can choose the most appropriate process trees guided by the quality scores for each process tree. For e.g., if the domain expert wants to focus on process trees with high fitness, then Pareto front could be navigated to only look at process trees with high fitness. However this approach is highly subjective and would depend entirely on the preference of the domain expert, and hence would be difficult to conduct in a scientific manner. Another approach for evaluation is to discover an *expected* model based on user specified constraints. In this

Table 8: Quality dimensions of the Pareto front for process trees from Figure 12

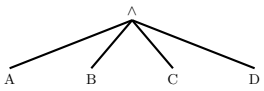
tree	constraints satisfied	replay fitness	precision	generalization	simplicity
Figure 12a	0	1	0.833	0.957	1
Figure 12b	4	0.997	1	0.957	1
Figure 12c	2	0.999	0.900	0.957	1
Figure 12d	2	0.998	0.993	0.957	1

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 ground truth that can be used to quantify results, and be used to evaluate the results in a controlled way without depending on the high subjectivity of the domain expert. We evaluate our approach based on both a synthetic log and a real life log. The synthetic log demonstrates the usage of our approach on small process trees. The real life event log is from a road traffic fine management process.

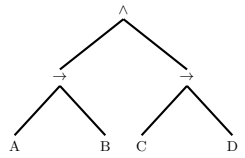
6.2 Synthetic Event Log

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) [17] with default settings generates the process tree with all four activities in parallel as shown in Figure 12a.

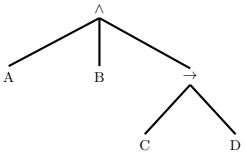
From the high frequent traces of the log we can deduce simple rules such as activity A is always eventually followed by activity B; and activity B is always preceded by activity A. Similar relationship holds for activities C and D. We use



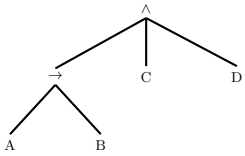
(a) Original process tree as discovered by Inductive Miner infrequent.



(b) Modified process tree present in the output of all modification approaches.



(c) Modified process tree present in the output of all modification approaches.



(d) Modified process tree present in the output of all modification approaches.

Fig. 12: Original and modified process trees for event log L.

Table 9: Performance statistics for the modification approaches from Section 5

approach	total # of trees discovered	total # of trees in Pareto front	average computation time for 5 runs (in ms)
Brute force	8323	88	53232
Genetic	50000[1]	112[2]	134689
Constraint specific	481	4	295

[1] *50000 is the maximum # of trees that could be produced (including duplicates)*
[2] *112 is the average number of trees in Pareto front over five runs*

this information to deduce the following four constraints: *response(A,B), precedence(A,B), response(C,D),* and *precedence(C,D).* We use these constraints, the process tree from Figure 12a discovered using IMi [17] and the synthetic log (L) as our input and perform the experiment with all three modification approaches discussed in Section 5. For the genetic modification approach, we set the stopping condition of maximum generations to 500 generations. The population size in each generation is set to 100. Along with this, we add some additional conditions restricting duplicate labels and guaranteeing minimum number of nodes. All the other settings are kept as default as in the ETM miner of [4]. In case of brute force modification, the maximum number of edits is set to 3.

Table 9 summarizes the performance statistics and output of each modification algorithm. The constraint specific modification algorithm outperforms the other approaches in terms of time taken, but also produces fewer process trees compared to the other two approaches. The Genetic algorithm produces a Pareto front containing varying number of process trees in each new run. Also, it takes the longest time as every process tree created in each generation is subject to alignments, which is often time consuming. The brute force modification is faster than the genetic modification as the number of nodes in the initial tree and the number of activities is very low. Figure 12 shows the original process tree discovered by IMi(Figure 12a) and 3 modified process trees present in the Pareto front output of each modification approach. Table 8 summarizes the dimension scores of the process trees from Figure 12.

The modified process tree from Figure 12b satisfies all the four constraints. The tree from Figure 12b also has a higher precision value of 1, and a considerably high replay fitness score. This process tree is highly precise, thereby explaining the high frequent traces of the event log much better and ignoring the infrequent noisy traces. Hence, if the user is interested the most in precision, then this is the process tree that would be chosen. However, if the user is a little relaxed w.r.t. to precision then process trees from Figure 12c and Figure 12d might be interesting. These process trees have a very high precision score along with very high fitness scores, outperforming the originally discovered process tree. However, only half the constrains specified are present in these process trees. Hence, if the user decides that some constraint(s) can be relaxed/ignored, then these process trees

Table 10: Domain rules derived from the complete real life event log.

Activity 1	Constraint	Activity 2
Insert Fine Notification	Precedence	**Send Appeal to Perfecture**
Receive Result Appeal from Perfecture	Responded Existence	Notify Result Appeal to Offender
Appeal to Judge	Not Succession	Send Appeal to Perfecture
Insert Date Appeal to Perfecture	Response	Send Appeal to Perfecture

Activities in **bold** are the primary activities.

could be chosen by the user. Based on user's preferences, the Pareto front could be navigated to select the most interesting process models.

6.3 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 activities and 150,370 cases available at [9]:

1. Use the complete event log to mine a process model using the heuristics miner. Learn *domain rules* based on this model and the log visualizers from ProM.
2. Filter the event log to select 1% of the cases having *exceptionally deviating* behavior.
3. Create a process tree based on the filtered log using inductive miner infrequent [16]. The BPMN representation of this process tree is shown in Figure 13.
4. Use the domain rules learned in step 1, the process tree from the filtered log of step 3, in combination with the *complete* event log as input to each of the modification approaches.

We deduce 1 precedence, 1 response, 1 responded-existence and 1 not-succesion rule from the complete event log using the heuristics miner and the log visualizers from ProM. These rules are shown in Table 10. It should be noted

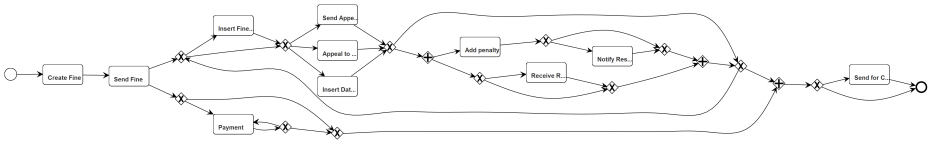


Fig. 13: BPMN model mined using IMi with filtered log containing infrequent traces only.(see appendix for bigger version.)

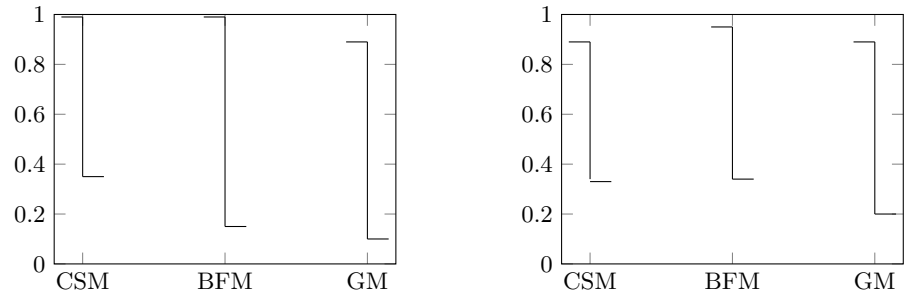
Table 11: Dimensions statistics for process trees based on real life event log

tree	constraints satisfied	replay fitness	precision	generalization	simplicity
IMi	0	0.74	0.52	0.84	1
Alpha	0	0.74	0.72	0.96	1
ILP	0	1	0.37	0.99	1
Constraint Specific	4	0.99	0.64	0.99	1
Genetic	4	0.82	0.71	0.98	1
Brute Force	4	0.83	0.68	0.99	1

that these rules may or may not hold in the *actual* real life process model corresponding to the event log. However, all these rules have a high support and confidence according to the event log, and hence we consider them to be true.

For the genetic modification approach, we set the stopping condition of maximum generations to 500 generations with the population size set to 100. All the other settings are kept as default from the ETM miner of [4]. Therefore, in total a maximum of 50000 trees are evaluated. In case of brute force modification, the maximum number of edits is set to 3. Furthermore, due to limited computing resources, the maximum number of trees selected for Pareto front evaluation in both brute force modification and constraint specific modification approach was limited to first 5000 trees satisfying at least 50% of the constraints. That is, for constraint specific modification approach and brute force modification approach, the total number of trees kept in memory were 5000 each, which were then used to populate the Pareto front.

In Table 11 we compare the evaluation outcomes of different approaches in terms of fitness, precision, generalization and simplicity. Since we considered the original complete event log as the ground truth, in the discussion to follow we primarily focus on the replay fitness and precision dimensions, as these two



(a) Range of fitness scores of all the trees from the Pareto front for each algorithm.

(b) Range of precision scores of all the trees from the Pareto front for each algorithm.

Fig. 14: Range of fitness and precision scores for process trees from Pareto front output based on real life event logs.

dimensions give a very good description of the model in terms of the event log. When the process model obtained from filtered event log (Figure 13), is evaluated against the complete event log, it features very poorly with very low fitness and precision scores of 0.74 and 0.52 respectively. Furthermore, we used a few of the automated process discovery algorithms to discover a model based on the filtered event log. Each of these models were then evaluated against the complete event log, and the results presented in the Table 11. Model mined by ILP miner returned a perfect fitness score of 1, even against the complete event log. However, many of the activities in the discovered model were completely isolated. Hence, it allowed for any behavior thereby fitting the complete log very well, but having extremely poor precision value of 0.34. The alpha algorithm generated a model which had the same score (of 0.74) for replay fitness as the IMi, and a much better precision value of 0.72.

As a next step we compare the outcomes of models mined using domain knowledge against the models mined without domain knowledge. For this we need to choose some models from the outputs of each of the modification approaches. All the three approaches contained at least 50 process trees in the Pareto fronts. As evident from Figure 14, the range of fitness and precision values is big, for process trees from the Pareto front. In order to compare the outcome of the modification approaches, we selected three process trees, one corresponding to each modification approach having a balanced high score of fitness and precision, and satisfying all the four constraints. All the three models chosen have a considerably higher fitness scored than the traditional techniques, except the model mined by ILP miner. Similarly, the precision scores of each of the selected model were usually considerably higher than the traditional process automated discovery techniques. This makes the improvements after adding the domain knowledge for post processing quite evident.

7 Conclusions and Future Work

In this paper we introduced a way to incorporate the domain knowledge of the expert in an already discovered process model. We primarily introduced two types of algorithms: verification and modification, in order to verify and incorporate domain knowledge in the process model respectively. 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. For modification, we have proposed three algorithms. The brute force approach exhaustively generates multiple process trees. However, the brute force approach does not consider the user constraints during the modification process. In order to overcome this, we introduce a genetic approach to consider both log and constraints during modification. As shown in Subsection 6.3, the guided genetic algorithm is considerably faster than the brute force approach when the number of activity classes is high. However, individually both brute force and genetic approach are time and memory inefficient. Hence, we

introduced another modification approach which changes the process model by individually considering each user constraint. Currently, underlying assumption of this approach is that the process model does not contain duplicate labels. Each modification approach has its own set of pro's and con's. For example, brute force modification and constraint specific modification techniques result in a consistent Pareto front, on repetitive runs of experiments. However, in the case of genetic approach, different runs could result in different process trees in the Pareto front. The constraint specific modification approach performs a pre-existing set of operations for modification and does not consider log information while modification, hence is faster than other approaches but rather limited in the number of process trees created. In the future, we would like 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] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst. Conformance checking using cost-based fitness analysis. In *Enterprise Distributed Object Computing Conference (EDOC), 2011 15th IEEE International*, pages 55–64. IEEE, 2011.

[2] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst. Towards robust conformance checking. In *Business Process Management Workshops*, volume 66 of *Lecture Notes in Business Information Processing*, pages 122–133. Springer Berlin Heidelberg, 2011.

[3] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verifiable agent interaction in abductive logic programming: The sciff framework. *ACM Trans. Comput. Logic*, 9(4):29:1–29:43, August 2008.

[4] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. A genetic algorithm for discovering process trees. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8. IEEE, 2012.

[5] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity. *Int. J. Cooperative Inf. Syst.*, 23(1), 2014. doi:10.1142/S0218843014400012.

[6] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. *Exploiting Inductive Logic Programming Techniques for Declarative Process Mining*, chapter Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems, pages 278–295. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-00899-3_16.

[7] C. Ciccio and M. Mecella. *Mining Constraints for Artful Processes*, chapter Business Information Systems: 15th International Conference, BIS 2012, Proceedings, pages 11–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-30359-3_2.

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

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

[10] P. M. Dixit, J. C. A. M. Buijs, W. M. P. van der Aalst, B. F. A. Hompes, and J. Buurman. Enhancing process mining results using domain knowledge.

[11] D. Fahland and W. M. P. van der Aalst. Repairing process models to reflect reality. In *Business process management*, pages 229–245. Springer, 2012.

[12] L. Giordano, A. Martelli, M. Spiotta, and D. T. Dupre. Business process verification with constraint temporal answer set programming. *Theory and Practice of Logic Programming*, 13:641–655, 7 2013. `doi:10.1017/S1471068413000409`.

[13] S. Goedertier, D. Martens, J. Vanthienen, and B. Baesens. Robust process discovery with artificial negative events. *J. Mach. Learn. Res.*, 10:1305–1340, June 2009.

[14] G. Greco, A. Guzzo, F. Lupa, and P. Luigi. Process discovery under precedence constraints. *ACM Trans. Knowl. Discov. Data*, 9(4):32:1–32:39, June 2015.

[15] E. Lamma, P. Mello, F. Riguzzi, and S. Storari. *Applying Inductive Logic Programming to Process Mining*, chapter Inductive Logic Programming: 17th International Conference, ILP 2007, pages 132–146. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. `doi:10.1007/978-3-540-78469-2_16`.

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

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

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

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

[20] E. Ramezani, D. Fahland, and W. M. P. van der Aalst. Where did I misbehave? diagnostic information in compliance checking. In *Business Process Management*, pages 262–278. Springer Berlin Heidelberg, 2012.

[21] Aubrey J Rembert, Amos Omokpo, Pietro Mazzoleni, and Richard T Goodwin. Process discovery using prior knowledge. In *Service-Oriented Computing*, pages 328–342. Springer, 2013.

[22] W. Runte and M. El Kharbili. Constraint checking for business process management. In *GI Jahrestagung*, pages 4093–4103, 2009.

[23] W. M. P. van der Aalst. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016.

[24] J. M. E. M. Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik. *Process Discovery Using Integer Linear Programming*, chapter Applications and Theory of Petri Nets: 29th International Conference, PETRI NETS 2008, Xi'an, China, June 23-27, 2008. Proceedings, pages 368–387. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi: 10.1007/978-3-540-68746-7_24.

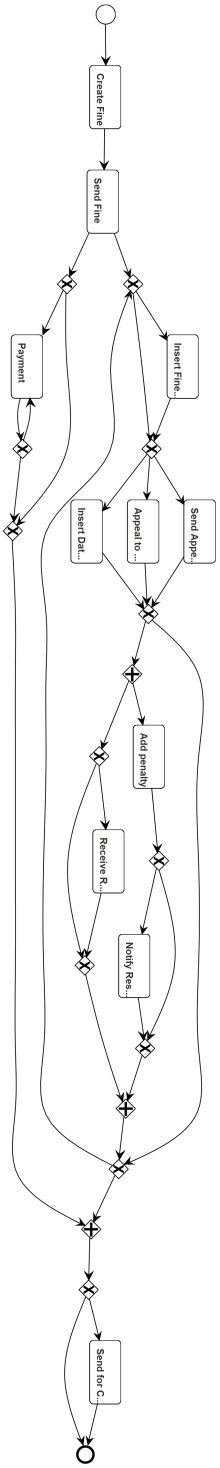


Fig. 15: BPMN model corresponding to the filtered real life event from Figure 13.