Mining Business Process Variants: Challenges, Scenarios, Algorithms

Chen Li[a], Manfred Reichert[b], Andreas Wombacher[c]

[a]*Information Systems Group, University of Twente, The Netherlands (lic@cs.utwente.nl) (✉)*
[b]*Institute of Databases and Information Systems, Ulm University, Germany (manfred.reichert@uni-ulm.de)*
[c]*Database Group, University of Twente, The Netherlands (a.wombacher@utwente.nl)*

Abstract

During the last decade a new generation of process-aware information systems has emerged, which enables process model configurations at buildtime as well as process instance changes during runtime. Respective adaptations result in a large number of process model variants that were derived from the same process model, but slightly differ in structure. Generally, such model variants are expensive to configure and maintain. This paper introduces two different scenarios for learning from process model adaptations and for discovering a reference model out of which the variants can be configured with minimum efforts. The first scenario presumes a reference process model and a collection of related process model variants. The goal is to evolve the reference process model such that it structurally fits better to the given variant models. The second scenario comprises a collection of process variants, while the original reference model is unknown; i.e., the goal is to "merge" these variants into a reference process model. We suggest two algorithms that are applicable in both scenarios, but which have their pros and cons. We systematically compare the two algorithms and contrast them with conventional process mining techniques. Our comparison results indicate good performance of both algorithms. Further they confirm that specific techniques are needed for learning from past process adaptations. Finally, we present a case study in the automotive industry in which we applied our algorithms.

1. Introduction

In today's dynamic business world success of an enterprise increasingly depends on its ability to react to environmental changes in a quick, flexible and cost-effective way [1, 2, 3]. To increase the flexibility of Process-Aware Information Systems (PAISs) different approaches for structurally adapting pre-modeled processes exist, e.g., by adding, deleting or moving process activities [4, 5]. Respective adaptations are not only needed at buildtime for customizing a given reference process model to a particular context [6, 7], but also become necessary for tailoring process instances during runtime in order to deal with exceptional situations and changing needs [8, 4]. As example consider medical guidelines for patient treatment processes [2], which need to be customized to fit to the particular healthcare environment in which they are used. Additional adaptations may become necessary when applying such tailored guideline to a particular patient [2]. Generally, respective adaptations lead to large collections of process model variants (*process variants* for short) derived from the same process model, but slightly differing in structure [1, 9]. In case studies we conducted in the healthcare domain and in automotive engineering, for example, we identified scenarios with dozens up to hundreds of variants [10].

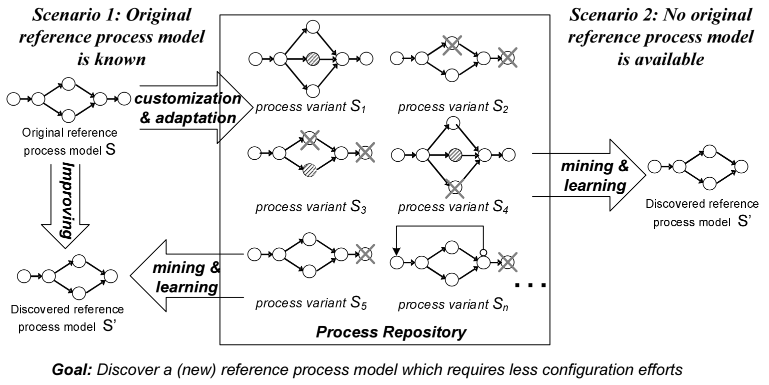

Scenario 1: Original reference process model is known
Original reference process model S
Improving
Discovered reference process model S'
customization & adaptation
mining & learning
process variant S_1
process variant S_2
process variant S_3
process variant S_4
process variant S_5
process variant S_n
Process Repository
Scenario 2: No original reference process model is available
mining & learning
Discovered reference process model S'
Goal: Discover a (new) reference process model which requires less configuration efforts

Figure 1: Different scenarios for discovering reference process models

1.1. Problem Statement

Though considerable efforts have been made to enable process configuration and adaptation [5, 6, 8, 7], most existing approaches have not utilized information about such structural adaptations yet [11]. Fig. 1 describes the overall goal of our research. We want to learn from past process model adaptations in order to discover a (new) *reference process model* covering the given variant collection best. By adopting the discovered model in the PAIS, need for future process adaptations and costs for change will decrease. Generally, finding such an improved reference model is by far not trivial when considering control flow patterns like sequence, parallel branching, conditional branching, and loops. Furthermore, major changes of the current reference process model might be not always preferred due to high implementation costs or for social reasons. Fig. 1 further differentiates between two fundamental scenarios. In the *first scenario* the process variants are derived by configuring a known reference process model. However, when discovering the new reference process model without considering the old one, we might be confronted with significant structural differences between old and new reference model. Process engineers should therefore have the flexibility to control to what degree they want to maximally modify the original reference model to better fit to the given variant collection. Consequently, closeness of the new reference model to the old one and closeness of this model to the given variant collection act as ”counterforces”. The *second scenario* we consider is based on a collection of related process variants, but does not presume any knowledge about the original reference process model these variants were derived from. Here we want to discover a reference process model by ”merging” these variants without considering the aforementioned ”counterforces” .

1.2. Contribution

Based on the assumptions that process models are block-structured [8, 12] and all activities in a process model have unique labels[1], this paper deals with the following *research questions*:

1. Given the original reference process model and a collection of related process variants, how can we derive a new reference process model that fits ”better” to these variants? — In this scenario we want to control the *evolution of the reference process model*, i.e., we want to enable process engineers to control to what degree the new reference model ”differs” from the original one and how ”close” it is to the variant collection.

[1]The block-structure constraint is discussed in Section 2. Regarding unique activity labeling, we refer to [13] for an approach that matches activities with different labels.

2. Given a collection of process variants without knowledge about the process model they were derived from, how can we discover a reference process model such that the *average distance* between this reference model and the process variants becomes minimal?
3. Which algorithms foster these two scenarios and what are their commonalities and differences? How do these algorithms differ from traditional process mining algorithms that focus on execution behavior?

As input for our analysis we solely require a collection of *process variant models* (and a *reference process model* in the first scenario). In particular, we do NOT presume the existence of *process change logs* which comprise information about the change patterns that were applied when configuring the variants out of the original reference model [14]. Furthermore we measure the closeness (or distance) between a reference process model and a related process variant in terms of the number of high-level change operations (e.g., to insert, delete or move activities [8]) needed to transform the reference process model into the respective variant. As reported in [15] the shorter this distance is, the less the efforts for configuring the variant (i.e., for structurally adapting the reference process model to derive the variant) are.

In the *first scenario*, we discover a new reference model by applying a sequence of change operations to the original one. Thereby, process engineers have the flexibility to control the similarity between original reference model and newly discovered one, i.e., they may specify how many change operations shall be maximally applied to the old reference model when discovering the new one. As benefit of this approach, we can control the efforts for evolving the reference process model. Further, we can avoid Spaghetti-like model structures — a common challenge in the field of process mining [16, 17]. Finally, in order to support the first scenario we target at an approach that considers changes, which significantly contribute to reduce the average distance between the discovered reference model and the given variant collection, first and less relevant ones last (cf. Section 4 for a detailed explanation). In the *second scenario*, we "merge" the variants without considering any original reference process model as "counterforce". Based on this simplification we provide another algorithm which shall perform better than the one designed for the first scenario. We systematically compare the two mining algorithms. We further compare them with existing process mining algorithms [18]. The latter aim at discovering a process model by analyzing the execution behavior of completed process instances as captured in execution logs [18, 19, 17, 20]. Respective logs typically document the start/end of each activity execution and therefore reflect the behavior of the implemented processes. In principle, process mining techniques can be applied in our context as well. However, they discover models which cover *behavior* best; i.e., their goal is different from ours.

This paper significantly extends our previous work presented in [21]. It handles more process patterns (e.g., loops), provides more technical and formal details, adds another mining algorithm, and discusses results from a case study. Finally, we compare our algorithms for process variants mining with existing process mining algorithms based on different criteria.

The remainder of this paper is organized as follows. Section 2 gives background information and introduces a running example. Section 3 deals with important measures for evaluating process variants in different aspects. Section 4 describes a heuristic algorithm we designed for Scenario 1, while Section 5 presents a clustering algorithm for handling Scenario 2. We compare the two algorithms with each other as well as with traditional process mining algorithms in Section 6. Results of a case study in the automotive domain are presented in Section 7. Section 8 discusses related work, while Section 9 concludes with a summary.

2. Backgrounds

A *process management system* (PrMS) provides generic process support functions and allows for separating process logic from application code. For this purpose, the process logic has to be explicitly defined based on the modeling patterns provided by a *process meta model*. At runtime the PrMS then orchestrates the processes according to the defined logic. For each business process to be supported, a *process type* represented by a *process model* S has to be defined. In this paper, a process model is represented as directed graph, which comprises a set of nodes - either representing *process steps* (i.e., *activities*) or control connectors (e.g, *And-/Xor-Split*) – and a set of *control edges* between them. The latter specify *precedence* as well as *loop backward relations*. Furthermore, we presume that process models are *block-structured*.

Fig. 2 depicts an example of such a process model. Nodes are represented as rectangles while precedence and loop backwards relations are expressed as directed edges of different type. Each process model contains a unique start and a unique end node.[2] For control flow modeling the following patterns are available: Sequence, AND-split, AND-join, XOR-split, XOR-join, and Loop [22]. These patterns constitute the core of any process specification language and cover most of the process models we can find in practice [23, 24]. Further, they can be easily mapped to other process execution languages like WS-BPEL as well as to formal languages like Petri Nets [25, 26]. Based on these patterns we are able to compose more complex process structures if required (e.g., an OR-split can be mapped to AND- and XOR-splits [27]).

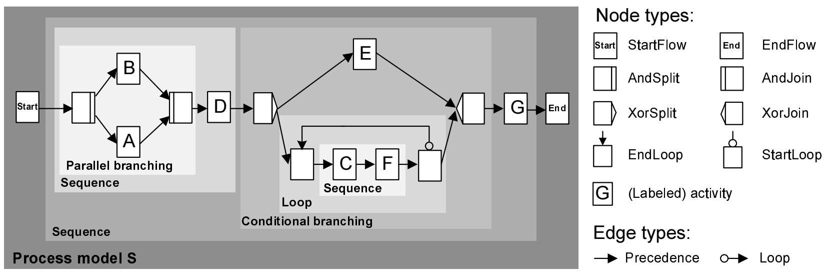


Figure 2: Example of a block-structured process model

Each node a of a process model S may have a label $l(a)$. Such labeled nodes constitute the *activities* of S (e.g., activities A and B in Fig. 2). Unlabeled nodes, in turn, represent silent activities. These have no associated actions and only exist for control flow purpose (e.g., the split and join nodes in Fig. 2) [28]. In the context our research, for each labeled activity we assume that its label is unique.[3] Regarding the example from Fig. 2, process model S contains 7 normal activities and 8 silent ones (including the start and end nodes).

We presume that a process model S is *block-structured*; i.e., activities, sequences, branchings, and loops constitute blocks (cf. Fig. 2) with unique *start* and *end* nodes[4] [29, 30, 31, 12]. These blocks may be nested, but must not overlap; i.e., their nesting must be regular [8, 12, 29, 31]. Generally a block can be a single activity, a sequence, a parallel branching, a conditional

[2]For the sake of readability, we omit the start and end node of a process model if its start and end are clear. For example, this applies to all process models from Fig. 4.

[3]Since activity labels are unique we assume that two activities from different process variant models are the same if they have same label. Otherwise, we refer to [13] for an approach that matches activities from different process models in case they have different labels.

[4]For example, in Fig. 2 the parallel block comprising activities A and B, and the subsequent activity D together constitute a block.

branching, or S itself. In Fig. 2, the grey areas show selected blocks in process model S, and the different grey levels indicate their nesting. In principle, we can consider a block itself as block-structured process model. In the following, we represent each block by a set of activities since the block-structure itself can be derived from the overall process model S; e.g., block {A,B} corresponds to the parallel block with AND-split and AND-join nodes in S. Similarly, {A}, {A,B,D}, {C,F}, and {A,B,C,D,E,F,G} describe selected blocks contained in S.[5] The concept of block-structuring has been known from block-structured programming languages for a long time [32], and can be found in process specification language like WS-BPEL and XLANG as well. Further, process management systems with block-structured process modeling language like AristaFlow BPM Suite [33] and CAKE2 [34] emerged, which have been applied to a variety of processes from different domains. When compared to unstructured process models, block-structured ones are easier understandable and have lower error probability [35, 36, 37]. If a process model is not block-structured, in many cases we can transform it into a block-structured one [31, 36, 12, 29]. For example, in a case study we analyzed 214 process models from different domains and expressed in different languages (e.g., Event Process Chains, UML Activity Diagrams). More than 95% of them were block-structured or could be transformed into a block-structured representation [38]. Despite the fact that there exist unstructured process models which cannot be transformed into block-structure, we consider our mining algorithms for block-structured process variant models as highly relevant.

Formally, we define *block-structured process model* and *block* as follows:

Definition 1 (Block-structured Process Model and Block).

1. *A tuple $S = (A, E, AT, ET, l)$ is called block-structured process model iff:*
   - *A is a set of `nodes` and AT assigns to each node $a \in A$ a `node type` $AT(a) \in$ {`StartFlow, EndFlow, Normal, AndSplit, AndJoin, XorSplit, XorJoin, StartLoop, EndLoop`}*
   - *$E \subseteq A \times A$ is a set of `directed edges` and ET assigns to each edge $e \in E$ an `edge type` $ET(e) \in$ {`Precedence,Loop`}. Further, $n \prec m :\Leftrightarrow n$ directly or indirectly precedes m when only considering edges of type `precedence`.*
   - *Let L be a set of activity labels. $l : A \rightarrow L$ is a partial `labeling function` which assigns a label $l(a) \in L$ to a node $a \in A$.*
   - *S has the block-structure properties as described above (for a formal and precise description see Appendix A).*
2. *Let $a_1, a_2 \in A$ with $a_1 \prec a_2$ or $a_1 = a_2$. Let further $join(a)$ be a bijective function to map each split/startLoop node $a \in A$ with $AT(a) \in$ {`AndSplit,XorSplit,StartLoop`} to its corresponding join/endLoop node $a' \in A$ with $AT(a') \in$ {`AndJoin,XorJoin,EndLoop`}. Then: The subgraph of S induced by node set $B = \{a_1, a_2\} \bigcup \{a \in A | a_1 \prec a \wedge a \prec a_2\}$ constitutes a **block** iff:*
   - *$\forall a \in B$ with $AT(a) \in$ {`AndSplit,XorSplit,StartLoop`}, $\Rightarrow join(a) \in B$*
   - *$\forall a \in B$ with $AT(a) \in$ {`AndJoin,XorJoin,EndLoop`}, $\Rightarrow join^{-1}(a) \in B$*

We do not provide an operational semantics for block-structured process models here, but refer to [8, 39, 29] instead. As the process patterns used in block-structured process model can

[5]Here, {C,F} only refers to the sequence structure containing activities C and F. We discuss in Section 3.1 how loop blocks are represented.

be easily mapped to WS-BPEL [25] or Petri Nets [22, 26], we could also describe the operational semantics of block-structured process models based on these languages. In principle, we can consider block-structured process models as a subclass of Workflow Nets [26], for which the net models follow respective structuring constraints. Similar to a Workflow Net, we consider a block-structured process model S as being *sound* iff the following properties hold: (1) *proper completion* (i.e., when the EndFlow node becomes enabled, all other nodes cannot be enabled anymore), (2) *absence of deadlocks* (i.e., as long as the EndFlow node has not been enabled, there is no execution situation in which no node is enabled), and (3) *absence of dead tasks* (i.e., there exists no node, which can be never enabled). For a formal description of these properties, we refer to [26, 40, 29]. We consider soundness as fundamental requirement any process model should satisfy as prerequisite for its proper execution and analysis [40, 8] (see [29, 8, 39, 40, 26] for techniques checking soundness). In the following, $\mathcal{P}$ denotes the set of all block-structured and sound process models.

Based on this, we define the notion of trace as follows:

Definition 2 (Trace). *Let $S = (A, E, AT, ET, l) \in \mathcal{P}$ be a sound and block-structured process model. Let further $t \equiv < a_1, a_2, \ldots, a_k >$ (with $a_i \in A$) be a sequence of activities. We denote t as trace of S iff:*

- *t is valid, i.e., the given execution sequence is producible on S.*
- *t is complete, i.e., a_1 is executed immediately after completion of the StartFlow node, and a_k is executed immediately before executing the EndFlow node.*

We define $\mathcal{T}_S$ as the set of all traces that can be produced by process model S.

We only consider traces that log events related to labeled activities, whereas events concerning silent activities are excluded (cf. Def. 1). As example consider process model S from Fig. 2. Sequences like `ABDEG`, `BADCFG` and `ABDCFCFCFG` constitute valid and complete traces of S. Like most process mining algorithms, we assume that the behavior of process model S can be expressed in terms of its trace set $\mathcal{T}_S$. Note that $\mathcal{T}_S$ can be an infinite set if S contains loops.

Process change: A process change is accomplished by applying a sequence of *high-level change operations* to the respective process model [8]. Such operations structurally modify a given process model by altering its set of activities and their order relations. The most relevant high-level change operations are *insert*, *delete*, and *move activity* as implemented in the ADEPT change framework [8]. Table 1 depicts these three change operations and informally describes their effects on process models. While *insert* and *delete* modify the activity set of a process

Change Operation Δ on S	opType	subject	paramList
insert(S, X, $\mathcal{A}$, $\mathcal{B}$, [sc])	insert	X	S, $\mathcal{A}$, $\mathcal{B}$, [sc]
Effects on S: inserts activity X between activity sets $\mathcal{A}$ and $\mathcal{B}$. `X` is conditionally inserted if [sc] is specified.			
delete(S, X)	delete	X	S
Effects on S: deletes activity X from S, i.e., X turns into a silent activity.			
move(S, X, $\mathcal{A}$, $\mathcal{B}$, [sc])	move	X	S, $\mathcal{A}$, $\mathcal{B}$, [sc]
Effects on S: moves activity X from its original position in S to another position between activity sets $\mathcal{A}$ and $\mathcal{B}$ (`X` is conditionally inserted if [sc] is specified).			

Table 1: *High-Level Change Operations*

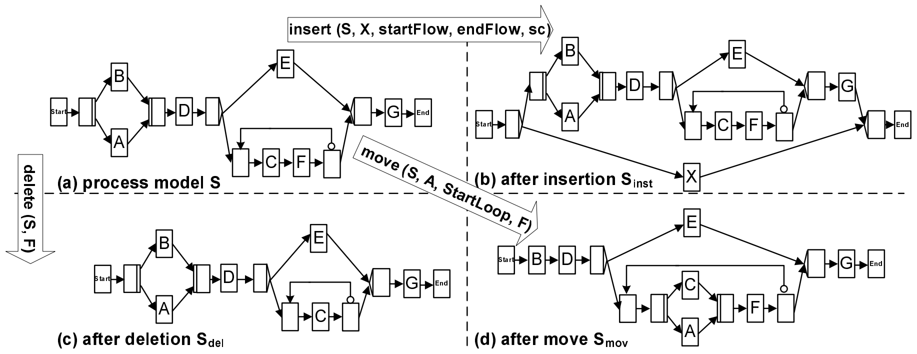


Figure 3: Influence of three change operations

model, *move* changes activity positions and thus process model structure (i.e., activity order relations). A formal semantics of these change operations is provided in [41]. It is based on the differences between the trace sets that can be produced on process models before and after their change. Issues concerning the correct use of change operations, their generalization and pre-/post-conditions are discussed in [8, 30]. Finally, by using high-level change operations instead of change primitives (i.e., elementary changes like adding or removing single nodes or edges), we can preserve block structure and guarantee soundness in the context of process changes. Further, we obtain a more meaningful measure for the distance between two models [28, 4].

As example consider Fig. 3. The original process model S is depicted in Fig. 3a. Fig. 3b - Fig. 3d depict process models S_{inst}, S_{del} and S_{mov} that result from S after performing an insertion, deletion or move operation. Note that silent activities, which only exist for control flow purpose, are adapted automatically when applying such high-level change operation. For example, when moving activity `A` within S (as expressed by operation *move*(S, `A`, `StartLoop`, `F`)) the two silent activities marking the AND-split and AND-join in S are removed automatically (cf. Fig. 3d).

Finally, a complex process change is accomplished by applying a sequence of high-level change operations to the given process model. An illustrating scenario is depicted in Fig. 4, where we list the change operations needed to transform model S into S_i ($i = 1, \ldots, 6$).

Definition 3 (Process Change and Process Variant). *Let $\mathcal{P}$ denote the set of sound and block-structured process models and let C be the set of possible process changes. Let $S, S' \in \mathcal{P}$ be two process models, let $\Delta \in C$ be a process change expressed in terms of a high level change operation, and let $\sigma = \langle \Delta_1, \Delta_2, \ldots \Delta_n \rangle \in C^*$ be a sequence of process changes performed on initial model S. Then:*

- *$S[\Delta\rangle S'$ iff Δ is applicable to S and S' is the (sound) process model resulting from the application of Δ to S.*
- *$S[\sigma\rangle S'$ iff $\exists\, S_1, S_2, \ldots S_{n+1} \in \mathcal{P}$ with $S = S_1$, $S' = S_{n+1}$, and $S_i[\Delta_i\rangle S_{i+1}$ for $i \in \{1, \ldots n\}$. We denote S' as **process variant** of S.*

Though the depicted change operations are discussed in relation to ADEPT, they are generic in the sense that they can be applied in connection with other process meta models as well [41, 4]; e.g., a process change as realized in ADEPT can be mapped to the concept of life-cycle inheritance known from Petri Nets [40]. We refer to ADEPT since it covers by far most high-level change patterns and change support features when compared to other adaptive PAIS [4]. Furthermore, with AristaFlow BPM Suite [33] an industrial-strength version of the ADEPT technology emerged, which has been extensively used in a variety of application domains [42].[6] Based on Def. 3 and the given change operations, we define *distance* and *bias* as follows:

Definition 4 (Bias and Distance). *Let S, $S' \in \mathcal{P}$ be two sound and block-structured process models. Then:* ***Distance*** *$d_{(S,S')}$ between S and S' corresponds to the minimal number of high-level change operations (cf. Table 1) needed to transform S into S'. We define $d_{(S,S')} = \min\{|\sigma| \mid \sigma \in C^* \wedge S[\sigma\rangle S'\}$. Furthermore, a sequence of change operations σ with $S[\sigma\rangle S'$ and $|\sigma| = d_{(S,S')}$ is denoted as* ***bias*** *$B_{(S,S')}$ between S and S'.*

The *distance* between process models S and S' corresponds to the minimal number of high-level change operations needed for transforming S into S'. The corresponding sequence of change operations is denoted as *bias* $B_{(S,S')}$ between S and S'.[7] Usually, distance measures the complexity of a process model configuration. As example take Fig. 4. Here, distance between model S and variant S_4 corresponds to *four*, since we minimally need to apply four change operations to transform S into S_4 [28]. In general, determining the bias and distance between two process models has complexity at $\mathcal{NP}$*-hard* level [28]. We refer to [28] for an approach that automatically computes the distance and bias between two block-structured models without need of a process change log.

Fig. 4 depicts an illustrating example of an original reference process model $S \in \mathcal{P}$ and 6 related process variants $S_i \in \mathcal{P}$ that were configured out of S by applying a sequence of change operations to it. All process models are block-structured (cf. Def. 1). Note that the variants do not only differ in structure, but also in respect to their activity sets; e.g., activity `X` appears in 5 of the 6 variants (except S_2), while `Z` only appears in S_5. Furthermore, variants may be weighted. In our context, we define the *weight* w_i of a process variant S_i as number of process instances that were created from S_i and executed on it; e.g., 25 instances were executed on S_1, while 20 instances ran on S_2. If we only know the variants, but have no runtime information about related instance executions, we assume variants to be equally weighted; then every variant has weight 1.

We can compute the distance (cf. Def. 4) between an original reference model S and each process variant S_i as well as related biases. For example, when comparing S with S_1 we obtain 5 as distance (cf. Fig. 4); i.e., we need to apply five high-level change operations to transform S into S_1: *delete*(*loop*), *move*(S, `H`, `I`, `D`), *move*(S, `I`, `J`, *endFlow*), *move*(S, `J`, `B`, *endFlow*), and *insert*(S, `X`, `E`, `B`) (cf. Def. 3). Based on the weight w_i of each variant S_i, we can compute *average weighted distance* between a reference model S and its variants as follows:

Definition 5 (Average Weighted Distance). *Let $S \in \mathcal{P}$ be a reference process model. Let further $\mathcal{M}$ be a set of process variants $S_i \in \mathcal{P}$, $i = 1, \ldots, n$, derived from S, with w_i representing the number of process instances that were executed on basis of S_i. The* ***Average Weighted Distance*** *$D_{(S,\mathcal{M})}$ between S and $\mathcal{M}$ can be computed as follows:*

$$D_{(S,\mathcal{M})} = \frac{\sum_{i=1}^{n} d_{(S,S_i)} \cdot w_i}{\sum_{i=1}^{n} w_i} \tag{1}$$

[6] Visit www.aristaflow-forum.de for more information and illustrating screen casts.

[7] Generally, it is possible that there exists more than one minimal sequence of change operations for transforming S into S', i.e., the bias of the two models does not need to be unique. Therefore, $B_{(S,S')}$ refers to an element from a set that contains all possible change sequences which transform S into S' and which comprise $d_{(S,S')}$ change operations. In this paper, we do not make such difference and for any σ with $S[\sigma\rangle S'$ and $|\sigma| = d_{(S,S')}$, we consider it as a bias (see [40, 28] for a detailed discussion of this issue).

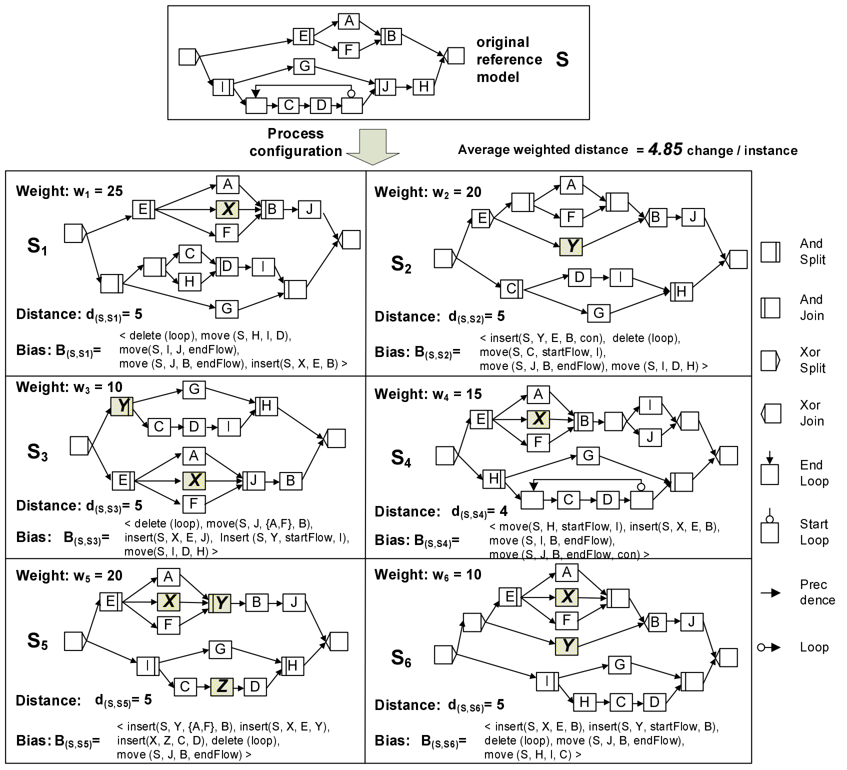

original reference model S
Process configuration
Average weighted distance = 4.85 change / instance
Weight: w1 = 25
S1
Distance: d(S,S1)= 5
Bias: B(S,S1)= < delete (loop), move (S, H, I, D), move(S, I, J, endFlow), move (S, J, B, endFlow), insert(S, X, E, B) >
Weight: w2 = 20
S2
Distance: d(S,S2)= 5
Bias: B(S,S2)= < insert(S, Y, E, B, con), delete (loop), move(S, C, startFlow, I), move (S, J, B, endFlow), move (S, I, D, H) >
Weight: w3 = 10
S3
Distance: d(S,S3)= 5
Bias: B(S,S3)= < delete (loop), move(S, J, {A,F}, B), insert(S, X, E, J), Insert (S, Y, startFlow, I), move(S, I, D, H) >
Weight: w4 = 15
S4
Distance: d(S,S4)= 4
Bias: B(S,S4)= < move(S, H, startFlow, I), insert(S, X, E, B), move (S, I, B, endFlow), move (S, J, B, endFlow, con) >
Weight: w5 = 20
S5
Distance: d(S,S5)= 5
Bias: B(S,S5)= < insert(S, Y, {A,F}, B), insert(S, X, E, Y), insert(X, Z, C, D), delete (loop), move (S, J, B, endFlow) >
Weight: w6 = 10
S6
Distance: d(S,S6)= 5
Bias: B(S,S6)= < insert(S, X, E, B), insert(S, Y, startFlow, B), delete (loop), move (S, J, B, endFlow), move (S, H, I, C) >
And Split
And Join
Xor Split
Xor Join
End Loop
Start Loop
Precdence
Loop

Figure 4: An illustrating example of a reference process model and related process variants

The complexity to compute average weighted distance is $\mathcal{NP}$*-hard* since the complexity to compute the distance between two variants is $\mathcal{NP}$*-hard* (cf. Def. 4). Regarding our example from Fig. 4, the distance between S and S_4 is 4, while the distances between S and S_i ($i \neq 4$) correspond to 5. When considering variant weights, we obtain as average weighted distance: $(5\times25+5\times20+5\times10+4\times15+5\times20+5\times10)/(25+20+10+15+20+10) = 4.85$. This means we need to perform on average 4.85 high-level change operations to configure a process variant S_i (and related instance respectively) out of reference process model S. Generally, average weighted distance between reference model and its variants expresses how "*close*" they are.

Our goal is to discover a reference model with shorter average weighted distance to a given collection of (weighted) process variants than the current reference model (Scenario 1), or minimal average weighted distance if the original reference model is unknown (Scenario 2).

3. Matrix-based Representation of Process Models and Process Variant Collections

As basic input for our mining algorithms we take a collection of process variants and optionally the original reference model they were derived from. This section shows how we represent this information. Its use will be discussed in Sections 4 and 5.

3.1. Representing a Block-structured Process Model as Order Matrix

This subsection first introduces *process structure trees* which provide a unique tree representation of block-structured process models [31] (cf. Section 3.1.1). Then we present the concept of *order matrix* which can be uniquely constructed out of a given process structure tree (cf. Section 3.1.3). Consequently, an order matrix also constitutes a unique representation of a block-structured process model. Representing process models in terms of a matrix is common in areas like process mining [17], process analysis [26] and process change management [28]. In particular, a matrix constitutes a mathematical representation which enables advanced analyses and processing options. In our mining algorithms (cf. Sections 4 and 5), we use order matrices as unique representations of block-structured process models.

3.1.1. Process Structure Tree

Transforming a block-structured model into a tree representation has its roots in *structured programming* and compiler theory [43]. Such transformation is applied, for example, when analyzing block-structured languages like XML or BPEL. In this paper we apply the approach from [31], which can transform a block-structured process model S into a *refined process structure tree* in linear time. Such refined process structure tree constitutes a unique representation of a process model. In the following, we denote it as *process structure tree* for short.

Definition 6 (Process Structure Tree). *A tuple $T = (N, C, CT, E, l)$ is called a process structure tree if the following holds:*

- *N is a set of* `nodes`*.*
- *C is a set of* `connectors` *and CT assigns to each connector $c \in C$ a* `connector type` *$CT(c) \in \{$*`Seq, AND, XOR, Loop`*$\}$.*
- *$E \subseteq (C \times C) \bigcup (C \times N)$ is a set of directed edges.*
- *For each $c_p \in C$ with $CT(c_p) =$* `Seq`*, its children in the process structure form an order $< a_{c_{p_1}}, a_{c_{p_2}}, \ldots, a_{c_{p_k}} >$ with $a_{c_{p_1}}, \ldots, a_{c_{p_k}} \in N \bigcup C$ and $(c_p, a_{c_{p_1}}), \ldots, (c_p, a_{c_{p_k}}) \in E$. We denote $a_{c_{p_i}} \ll_{c_p} a_{c_{p_j}}$ iff $i < j$.*
- *Let L be a set of activity labels, $l : N \rightarrow L$ is a partial* `labeling function` *which assigns a label $l(n)$ to a node $n \in N$.*

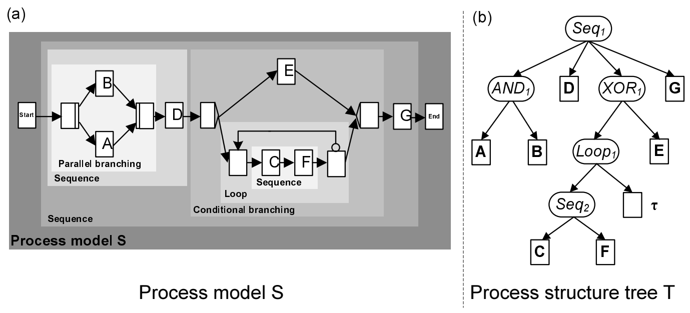


Figure 5: Process Model S and its corresponding process structure tree T

A process structure tree is an ordered tree which consists of a set of nodes, a set of connectors, and a set of directed edges linking them. The labeling function l assigns labels to nodes in a similar way as in Definition 1. Fig. 5 shows a block-structured process model S and its corresponding process structure tree T. In such a tree, nodes (represented as rectangles) correspond to activities while connectors (represented as ellipses) represent their relations based on process patterns like Sequence, AND-block, XOR-block, and Loop [22]. The precedence relations (expressed by connector `Seq`) are parsed from left to right; e.g., the AND-block containing activities `A` and `B` as well as their connector `AND`$_1$ precedes activity `D` since this block is on the left. Note that when representing a loop structure within a process structure tree T, we introduce a silent activity τ as direct successor of the respective `Loop` connector (cf. Fig.

5b). This way we ensure that any connector in T always has at least two successors.[8] In a process structure tree, nodes correspond to leaves, while connectors are non-leaves. Finally, a process structure tree has a unique root node without incoming edges.

Definition 7 (Ancestor and Subtree). *Let $T = (N, C, CT, E, l)$ be a process structure tree. Let $a, b \in N \bigcup C$ be two elements of T. Then:*

1. *a is an* ***ancestor*** *of b ($a \prec b$) $:\Leftrightarrow$ There exists a path from a to b.*
2. *$\mathcal{A}_T(a) = \{b | (b \in N \bigcup C) \wedge (a \prec b)\}$ is denoted as descendant set of element $a \in N \bigcup C$.*
3. *A* ***subtree*** *T' of tree T is a process structure tree $T' = (N', C', CT', E', l)$ with the following properties:*
    - *$(N' \subseteq N) \wedge (C' \subseteq C)$*
    - *$\exists a \in N' \bigcup C' : N' \bigcup C' = \mathcal{A}_T(a) \bigcup \{a\}$; i.e., a is the root element of T'*
    - *$E' = \{(a, b) | a, b \in N' \bigcup C' \wedge (a, b) \in E\}$*

A sub-tree $T' = (N', C', CT', E', l)$ of process structure tree $T = (N, C, CT, E, l)$ is a connected fragment of T which contains a unique root element $a \in N \bigcup C$ and all its descendants. As example consider Fig. 5b: Activities `A` and `B`, their connector `AND`$_1$, and the edges linking them can form a subtree of T. Since the process model S represented by T is block-structured, any subtree of T corresponds to a block of S; i.e., a process structure tree and its hierarchical decomposition into subtrees correspond to a block-structured process model and its hierarchical decomposition into blocks [31]. Given an element $e \in N \bigcup C$ in a process structure tree and taking Def. 7, we are able to construct a subtree $T(e)$ by identifying its descendant set $\mathcal{A}_T(e)$ and the edges linking them. In our example, the subtree of connector `Loop`$_1$ is a tree containing nodes `C,F` and τ, connectors `Loop`$_1$ and `Seq`, $_2$ and the edges connecting these elements.

The main reason to transform a block-structured process model into its corresponding process tree is as follows. A process structure tree contains less unnecessary silent activities such that it provides a clear picture of the process model's structure and the relations between its activities.

3.1.2. Nearest Common Ancestor

Before we apply process structure trees in our context, we introduce the concept of *nearest common ancestor*:

Definition 8 (Nearest Common Ancestor). *Let $T = (N, C, CT, E, l)$ be a process structure tree and let $a, b \in N$ be two different nodes of T. Then: we denote connector $c \in C$ as nearest common ancestor $NCA(a, b)$ of these two nodes iff:*

- *$c \prec a$ and $c \prec b$*
- *$\nexists c' \in C : c' \prec a$, $c' \prec b$ and $c \prec c'$*

Note that the nearest common ancestor of two nodes always refers to a connector since nodes constitute leaves of the process structure tree and consequently cannot be ancestors. Here we have assumed that a process structure tree contains at least two nodes and one connector. Finding the nearest common ancestor in a tree is a well researched topic. Based on the algorithms presented in [44], we are able to compute the nearest common ancestor for any two nodes in a process tree T in linear time; i.e., $O(n)$ with $n = |N \bigcup C|$.

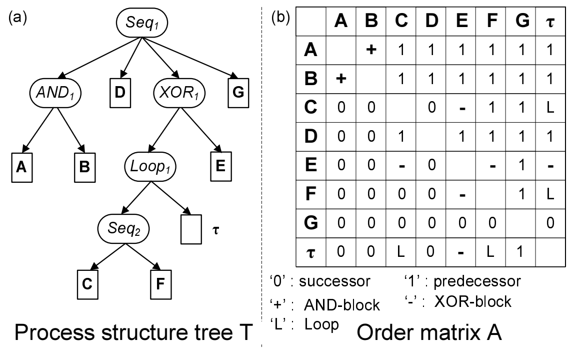
[8]Connectors of type `Seq`, `AND` and `XOR` represent a relation between its successors and require at least two successors. In case of "empty" branches in an XOR branching, silent nodes are introduced for representing them.

3.1.3. Representing a Process Model as Order Matrix

Based on the process structure tree T of a block-structured process model S and the concept of nearest common ancestor (cf. Def. 8), we can introduce the notion of *order matrix*, which uniquely represents a process structure tree and thus a block-structured process model.

Definition 9 (Order matrix). *Let S be a block-structured process model and let $T = (N, C, CT, E, l)$ be its process structure tree. A is called* ***order matrix*** *of T with $A_{a_i a_j}$ representing the order relation between activities a_i,$a_j \in N$, $i \neq j$ iff:*

- *$A_{a_i a_j}$ = '1' iff in T, a_i and a_j have as nearest common ancestor $c \in C$ with $CT(c) = \mathtt{Seq}$; let e_l and e_r be two children of c with $e_l \ll_c e_r$ and let further T_{e_l} and T_{e_r} be two subtree of T with root elements being e_l and e_r, a_i is contained in the T_{e_l} and a_j is contained in T_{e_r}.*
- *$A_{a_i a_j}$ = '0' iff in T, a_i and a_j have as nearest common ancestor $c \in C$ with $CT(c) = \mathtt{Seq}$; let e_l and e_r be two children of c with $e_l \ll_c e_r$ and let further T_{e_l} and T_{e_r} be two subtree of T with root elements being e_l and e_r, a_i is contained in the T_{e_r} and a_j is contained in T_{e_l}.*
- *$A_{a_i a_j}$ = '+' iff in T, a_i and a_j have as nearest common ancestor $c \in C$ with $CT(c) = \mathtt{AND}$.*
- *$A_{a_i a_j}$ = '-' iff in T, a_i and a_j have as nearest common ancestor $c \in C$ with $CT(c) = \mathtt{XOR}$.*
- *$A_{a_i a_j}$ = 'L' iff in T, a_i and a_j have as nearest common ancestor $c \in C$ with $CT(c) = \mathtt{Loop}$.*



	A	B	C	D	E	F	G	τ
A		+	1	1	1	1	1	1
B	+		1	1	1	1	1	1
C	0	0		0	-	1	1	L
D	0	0	1		1	1	1	1
E	0	0	-	0		-	1	-
F	0	0	0	0	-		1	L
G	0	0	0	0	0	0		0
τ	0	0	L	0	-	L	1	

'0' : successor '1' : predecessor
'+' : AND-block '-' : XOR-block
'L' : Loop

Figure 6: Process structure tree T and its corresponding order matrix A

Fig. 6 depicts the process structure T from Fig. 5b and its corresponding order matrix A. Note that silent activity τ, which was introduced as the direct successor of connector $\mathtt{Loop}_1$ in T, is included in the order matrix A as well (cf. Section 3.1.1). This order matrix contains all five order relations from Def. 9. For example, activities E and C have as nearest common ancestor connector $\mathtt{XOR}_1$. Thus, we assign '-' to matrix elements $A_{\mathtt{EC}}$ and $A_{\mathtt{CE}}$. Since activities B and G have as nearest common ancestor connector $\mathtt{Seq}_1$, and B is on its left subtree while G is on its right, we further obtain order relations $A_{\mathtt{BG}}$ = '1' and $A_{\mathtt{GB}}$ = '0' respectively. Special attention should be paid to the order relations between silent activity τ and the other activities. Since τ is direct successor of connector $\mathtt{Loop}_1$, order relation 'L', indicates those nodes in T which are descendants of this loop connector. Consequently, the order relations between τ on the one hand and activities C and F on the other hand are set to 'L'. This implies that C and F are descendants of a loop connecter and consequently implies that they are included in a loop block in the corresponding process structure tree and process model respectively. Note that the main diagonal of an order matrix is empty since we do not compute the nearest common ancestor of an activity with itself. Theorem 1 states that an order matrix A can uniquely represent a corresponding process structure tree T.

Theorem 1. *Let $T = (N, C, CT, E, l)$ be a process structure tree. Let further $A_{|N|\times|N|}$ be the order matrix constructed based on T. Then: Such order matrix A exists and is unique.*

Proof. See Appendix B.

Since a process structure tree T constitutes a unique representation of a block-structured process model S [31], and T can be uniquely represented by an order matrix A (cf. Theorem 1), A is a unique representation of S as well. Consequently, it is sufficient to analyze the order matrix of a block-structured process model. We make use of this in Sections 4 and 5. In [45] we provided algorithms which transform a process model directly to its order matrix and vice versa.

3.2. Representing a Collection of Process Variants as Aggregated Order Matrix

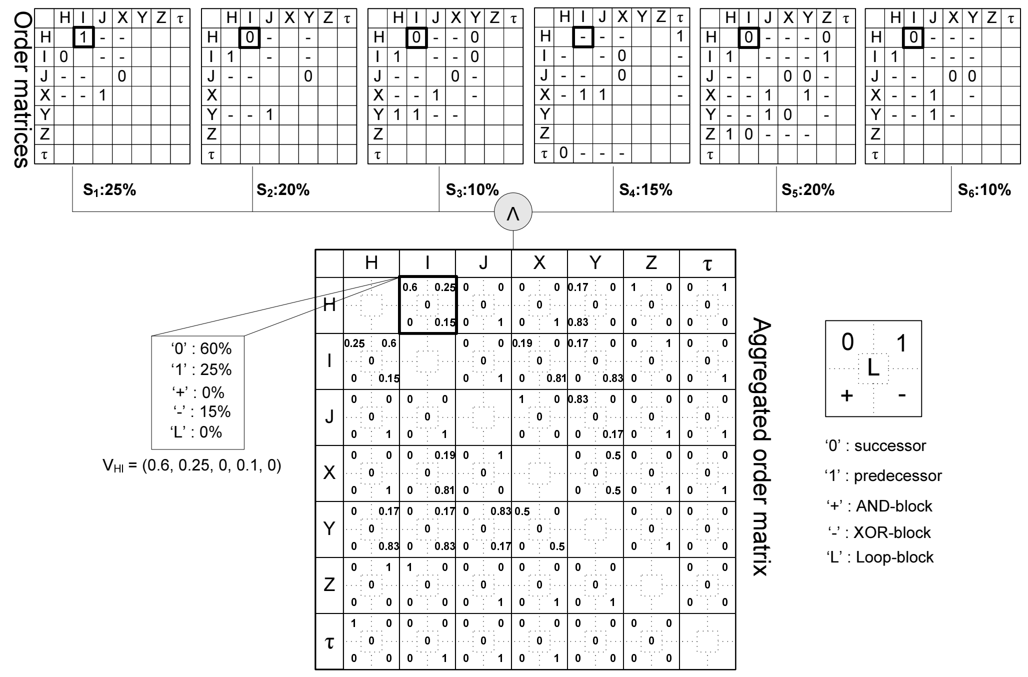


Figure 7: Aggregated order matrix based on process variants

In order to facilitate the analysis of a collection of process model variants, this section introduces the concept of *Aggregated Order Matrix*, which represents a collection of process model variants as a high-dimensional matrix. For computing the aggregated order matrix, we first determine the order matrix of each block-structured process variant. In this context, we consider activities from different variants being the same if they have the same label.[9] Regarding our example from Fig. 4, we need to compute six order matrices (cf. Fig. 7). Due to space limitations, Fig. 7 only provides a partial view on them (i.e., activities `H,I,J,X,Y,Z` and silent activity τ representing the Loop-block). Following this, we analyze the order relations for each pair of activities based on all derived order matrices. As the order relations between two activities might be not the same in all order matrices, this analysis does not result in a fixed relation, but in a distribution for the five types of order relations (cf. Def. 9). Regarding our example, in 60% of

[9]We refer to [13] for an approach that matches activities from different process models in case they have different labels. Note that we can also use this technique to handle silent activities which represent the loop structures in a process structure tree. If there are multiple silent activities in each of the process structure trees we can map these silent activities based on their context (e.g., their relationship to other activities). In the following, we assume that such mapping between activities (including silent ones) in different process structure trees has already been established.

all cases H succeeds I (as in S_2, S_3, S_5 and S_6), in 25% of all cases H precedes I (as in S_1), and in 15% of all cases H and I are contained in different branches of an XOR-block (as in S_4) (cf. Fig. 7). Generally, for a given variant collection we define the order relation between two activities a and b as 5-dimensional vector $V_{ab} = (v^0_{ab}, v^1_{ab}, v^+_{ab}, v^-_{ab}, v^L_{ab})$. Each vector field corresponds to the relative frequency of the respective relation type ('0', '1', '+', '-', or 'L') as specified in Def. 9. Take our example from Fig. 4 and consider Fig. 7: $v^1_{\mathtt{HI}} = 0.25$ corresponds to the frequency of all order matrices with activities H and I having order relationship '1', i.e., all cases for which H precedes I. We obtain $V_{\mathtt{HI}} = (0.6, 0.25, 0, 0.15, 0)$.

Definition 10 (Aggregated Order Matrix). *Let $S_i \in \mathcal{P}$, $i = 1, 2, \ldots, n$ be a collection of process variants. Let further $T_i = (N_i, C_i, CT_i, E_i, l_i)$ and A_i be the process structure tree and the order matrix of S_i, and w_i be the number of process instances that were executed on S_i. The **Aggregated Order Matrix** of all process variants is defined as 2-dimensional matrix $V_{m\times m}$ with $m = |\bigcup N_i|$ and each matrix element $v_{a_j a_k} = (v^0_{a_j a_k}, v^1_{a_j a_k}, v^+_{a_j a_k}, v^-_{a_j a_k}, v^L_{a_j a_k})$ being a 5-dimensional vector. For $\diamond \in \{0, 1, +, -, L\}$, element $v^{\diamond}_{a_j a_k}$ expresses to what percentage, activities a_j and a_k have order relation $\diamond$ within the given variant collection $S_1, \ldots, S_n$. Formally: $\forall a_j, a_k \in \bigcup N_i, a_j \neq a_k$:*

$$v^{\diamond}_{a_j a_k} = \frac{\sum_{A_{i_{a_j a_k}} = '\diamond'} w_i}{\sum_{a_j, a_k \in N_i} w_i}. \tag{2}$$

Fig. 7 partially shows the aggregated order matrix V for the process variants from Fig. 4. Due to space limitations, we only consider order relations for activities H, I, J, X, Y, Z, and silent activity τ which represents the Loop-block.

3.3. Measuring Activity Frequencies in a Variant Collection

Generally, the order relations computed by an aggregated order matrix may be not equally important. For example, relation $V_{\mathtt{HI}}$ between H and I (cf. Fig. 7) is more important than relation $V_{\mathtt{HZ}}$, since H and I co-appear in all six process variants, while H and Z only co-occur in S_5 (cf. Fig. 4). We introduce the *co-existence matrix CE* to indicate the importance of the different order relations that occur within an aggregated order matrix V.

	H	I	J	X	Y	Z	τ
H		1	1	0.8	0.6	0.2	0.15
I	1		1	0.8	0.6	0.2	0.15
J	1	1		0.8	0.6	0.2	0.15
X	0.8	0.8	0.8		0.4	0.2	0.15
Y	0.6	0.6	0.6	0.4		0.2	0
Z	0.2	0.2	0.2	0.2	0.2		0
τ	0.15	0.15	0.15	0.15	0	0	

Figure 8: Coexistence Matrix

Definition 11 (Coexistence Matrix). *Let $S_i \in \mathcal{P}$, $i = 1, 2, \ldots, n$ be a collection of process variants. Let further $T_i = (N_i, C_i, CT_i, E_i, l_i)$ be the process structure tree of S_i, A_i be the corresponding order matrix of S_i, and w_i be the number of process instances that were executed on S_i. The **Coexistence Matrix** of variant collection $\{S_1, \ldots, S_n\}$ is then defined as 2-dimensional matrix $CE_{m\times m}$ with $m = |\bigcup N_i|$. Each matrix element $CE_{a_j a_k}$ corresponds to the relative frequency with which activities a_j and a_k co-occur within the given variant collection. Formally: $\forall a_j, a_k \in \bigcup N_i, a_j \neq a_k$:*

$$CE_{a_j a_k} = \frac{\sum_{S_i : a_j, a_k \in N_i} w_i}{\sum_{i=1}^{n} w_i} \tag{3}$$

Fig. 8 shows the coexistence matrix for our running example. Again, we only depict the coexistence matrix for activities H, I, J, X, Y, Z, and silent activity τ; e.g., we obtain $CE_{\mathtt{HI}} = 1$ and $CE_{\mathtt{HZ}} = 0.2$. This indicates that the order relation between H and I is more important than the one between H and Z. For a given variants collection, we can further measure how frequent each activity a_i appears using *Activity Frequency*:

Definition 12 (Activity Frequency). *Let $S_i \in \mathcal{P}$, $i = 1, 2, \ldots, n$ be a collection of process variants. Let further $T_i = (N_i, C_i, CT_i, E_i, l_i)$ be the process structure tree of S_i, A_i be the corresponding order matrix, and w_i be the number of process instances that were executed on S_i. For each $a_j \in \bigcup_{i=1}^{n} N_i$, we define $g(a_j)$ as relative frequency with which a_j appears within the given variant collection. Formally:*

$$g(a_j) = \frac{\sum_{S_i : a_j \in N_i} w_i}{\sum_{i=1}^{n} w_i} \tag{4}$$

Table 2 shows the relative frequency of activities contained in the process variants of our running example (cf. Fig. 4); e.g., activity X is present in 80% of the variants (i.e., in S_1, S_3, S_4, S_5, and S_6), while Z only occurs in S_5 (i.e., 20% of the variants). Since S_4 contains a loop-block, we obtain 15% as the frequency with which silent activity τ occurs (cf. Def. 9).

Activity	A	B	C	D	E	F	G	H	I	J	X	Y	Z	τ
$g(a_j)$	1	1	1	1	1	1	1	1	1	1	0.8	0.6	0.2	0.15

Table 2: Relative frequency of each activity within the given variant collection

4. Scenario 1: Evolving Reference Process Models by Learning from Past Model Adaptations: A Heuristic Approach

As discussed in Section 2, measuring the distance between two block-structured process models (cf. Def. 4) constitutes an *$\mathcal{NP}$-hard* problem; i.e., the time for computing distance is exponential to the size of the process models. Consequently, the problem set out in our research question (i.e., finding a reference process model with minimal average weighted distance to the process variants) constitutes an *$\mathcal{NP}$-hard* problem as well. When encountering real-life cases (i.e., dozens up to hundreds of variants with complex structure), finding "the optimum" would either be too time-consuming or simply be not feasible. In this section, we present a *heuristic search algorithm* for mining process variants, while being able to control the maximum distance between old and new reference process model (cf. Scenario 1).

Heuristic algorithms have been widely used in fields like Artificial Intelligence [46], Data Mining [47] and Machine Learning [48]. A problem employs heuristics when "it may have an exact solution, but the computational cost of finding it may be prohibitive" [46]. Although heuristic algorithms do not aim at finding the "real optimum" (i.e., it is neither possible to theoretically prove that the discovered result is the optimum nor can we state how close it is to the optimum), they are widely used in practice. Usually, heuristic algorithms provide a nice balance between goodness of the discovered solution and computation time needed for finding it [46]. Informally, our *heuristic algorithm for process variants mining* works as follows:

Step 1.	Use the original reference model S as starting point.
Step 2.	Search for all neighboring process models with distance 1 to the currently considered reference process model S. If we are able to find a better model S' among these candidate models (i.e., one which is expected to have lower average weighted distance to the given variant collection when compared to S), we replace S by S'.
Step 3.	Repeat Step 2 until we either cannot find a better model or the maximally allowed distance between original and new reference process model is reached. The last S' then represents the discovered reference model.

Very important for any heuristic search algorithm are two aspects: the *heuristic measure* and the *algorithm* that uses heuristics to search the state space. Section 4.1 introduces the *fitness* *function* we suggest for measuring the "quality" of a particular candidate model. Section 4.2 then introduces a *best-first search* algorithm for searching the state space which contains all candidate process models.

4.1. Fitness Function

Generally, the fitness function of a heuristic search algorithm should be quickly computable. Since search space often becomes very large, we should be able to make a quick decision when performing the search. In our context, *average weighted distance* (cf. Def. 5) would be not a good choice since the complexity for computing it is $\mathcal{NP}$*-hard*. Therefore we introduce a *fitness function* which can be used to approximately measure "closeness" between a candidate reference model and the given variant collection. In particular, this fitness can be computed in polynomial time. Like in most heuristic search algorithms, the chosen fitness function is a "reasonable guessing" rather than a precise measurement. Section 4.4 evaluates our choice by investigating the correlation between our fitness function and average weighted distance (cf. Def. 5).

4.1.1. Activity Coverage

Given a candidate reference process model $S_c \in \mathcal{P}$ and its process structure tree $T = (N_c, C_c, CT_c, E_c, l_c)$ we first measure to what degree activity set N_c covers the activities that occur in the variant collection. Note that we also consider silent activities τ_k representing `loop` connectors in T (cf. Section 3.1.1). We denote this measure as *activity coverage* $AC(S_c)$ of S_c.

Definition 13 (Activity coverage). *Let S_i, $i = 1, \ldots, n$ be a collection of process variants, and let $T_i = (N_i, C_i, CT_i, E_i, l_i)$ be the process structure tree of S_i. Let further $M = \bigcup_{i=1}^{n} N_i$ be the set of activities that are present in at least one of the process structure trees. Let further $T_c = (N_c, C_c, CT_c, E_c, l_c)$ be the process structure tree of candidate process model S_c. Given activity frequency $g(a_j)$, for each $a_j \in M$ the* ***activity coverage*** *$AC(S_c)$ of S_c is defined as follows:*

$$AC(S_c) = \frac{\sum_{a_j \in N_c} g(a_j)}{\sum_{a_j \in M} g(a_j)} \tag{5}$$

Obviously, $AC(S_c) \in [0, 1]$ holds. Consider Fig. 4 and take the original reference model S as candidate model. It contains activities `A, B, C, D, E, F, G, H, I, J`, and τ (which represents the Loop-block). Its activity coverage $AC(S)$ expresses to what degree S covers the activities in the given variant collection; we obtain $AC(S) = \frac{10.15}{11.8} = 0.860$.

4.1.2. Structure Fitting of a Candidate Process Models

$AC(S_c)$ measures how representative the activity set of candidate model S_c is in respect to the given variant collection. However, it does not state how well the structure of S_c (i.e., its order relations) fits to these variants. We therefore introduce *structure fitting $SF(S_c)$* as second metrics. It measures to what degree S_c structurally fits to the given variants collection. For this purpose, we use the *aggregated order matrix* (cf. Def. 10) and *coexistence matrix* (cf. Def. 11).

Since we can represent a candidate process model S_c by its corresponding order matrix A_c (cf. Def. 9), we determine the structure fitting $SF(S_c)$ between S_c and the variants by measuring how similar the order matrix A_c and the aggregated order matrix V (representing the variants) are. Take original reference model S in Fig. 4 as candidate process model S_c (i.e., $S_c := S$). Obviously, $A_{\mathtt{HI}}$ ='0' holds, i.e., `H` succeeds `I` in S (cf. Fig. 4). Consider now the aggregated order matrix V representing the variants (cf. Fig. 7). Here the order relation between `H` and `I` is represented by the 5-dimensional vector $V_{\mathtt{HI}} = (0.6, 0.25, 0, 0.15, 0)$. If we now want to compare how close $A_{\mathtt{HI}}$ and $V_{\mathtt{HI}}$ are, we first need to build an aggregated order matrix V^c purely based on our candidate process model S_c (S in our case). Trivially, as order relation between H and I in V^c, we obtain $V^c_{\mathtt{HI}} = (1, 0, 0, 0, 0)$. We then compare $V_{\mathtt{HI}}$ (which represents the variants) with $V^c_{\mathtt{HI}}$ (which represents the reference model). We use Euclidean metrics $f(\alpha,\beta)$ to measure closeness between two vectors $\alpha = (x_1, x_2, ..., x_n)$ and $\beta = (y_1, y_2, ..., y_n)$:

$$f(\alpha,\beta) = \frac{\alpha \cdot \beta}{|\alpha| \cdot |\beta|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \cdot \sqrt{\sum_{i=1}^{n} y_i^2}} \in [0, 1] \tag{6}$$

$f(\alpha,\beta)$ computes the cosine value of the angle θ between vectors α and β in Euclidean space. If $f(\alpha,\beta) = 1$ holds, α and β exactly match in their directions; $f(\alpha,\beta) = 0$ means, they do not match at all. Regarding our running example, we obtain $f(V_{\mathtt{HI}}, V^c_{\mathtt{HI}}) = 0.899$. This indicates high similarity between the order relation of H and I in the candidate process model with the ones captured by the variants. Based on Euclidean metrics, which measures *similarity* between the order relations, and Coexistence matrix CE (cf. Def. 11), which measures *importance* of the order relations, we formally define structure fitting $SF(S_c)$ of a candidate model S_c as follows:

Definition 14 (Structure Fitting). *Let $S_i \in \mathcal{P}$, $i = 1, 2, \ldots, n$ be a collection of process variants and let $T_i = (N_i, C_i, CT_i, E_i, l_i)$ be the corresponding process structure trees. Let further CE be the coexistence matrix and V be the aggregated order matrix of this variant collection. For candidate model S_c, let $T_c = (N_c, C_c, CT_c, E_c, l_c)$ be the corresponding process structure tree, and let $m = |N_c|$ correspond to the number of nodes in T_c. Finally, let V^c be the aggregated order matrix of S_c. Then **structure fitting** $SF(S_c)$ is defined as follows:*

$$SF(S_c) = \frac{\sum_{j=1}^{m} \sum_{k=1, k \neq j}^{m} (f(V_{a_j a_k}, V^c_{a_j a_k}) \cdot CE_{a_j a_k})}{m \cdot (m-1)} \quad \in [0, 1] \tag{7}$$

For every pair of activities $a_j, a_k \in N_c, j \neq k$, we first compute the similarity of their corresponding order relations (as captured by V and V_c) in terms of $f(V_{a_j a_k}, V^c_{a_j a_k})$. Second, we determine the importance of these order relations by calculating $CE_{a_j a_k}$. Structure fitting $SF(S_c)$ of candidate model S_c then equals the average of the similarities multiplied with the importance of every order relation. Regarding our example from Fig. 4, we obtain $SF(S) = 0.632$ when choosing S as candidate model.

4.1.3. Fitness Function

Based on activity coverage $AC(S_c)$ (cf. Def. 13) and structure fitting $SF(S_c)$ (cf. Def. 14), we compute fitness $Fit(S_c)$ of a candidate model S_c as follows:

Definition 15 (Fitness). *Let $AC(S_c)$ be activity coverage of candidate model S_c and $SF(S_c)$ be its structure fitting. **Fitness** $Fit(S_c)$ of S_c is defined as follows: $Fit(S_c) = AC(S_c) \cdot SF(S_c)$*

As $AC(S_c) \in [0, 1]$ and $SF(S_c) \in [0, 1]$ hold, $Fit(S_c) \in [0,1]$ holds as well. $Fit(S_c)$ indicates how "close" candidate model S_c is to the given variant collection. If $Fit(S_c) = 1$, S_c will perfectly fit to the variants; i.e., no further adaptation will be needed. Generally, the higher $Fit(S_c)$ is, the closer S_c is to the variants and the less configuration efforts are required. In our example from Fig. 4, original reference model S has $Fit(S) = AC(S) \cdot SF(S) = 0.860 \cdot 0.632 = 0.543$. As the fitness of candidate model S_c is evaluated by multiplying activity coverage $AC(S_c)$ with structure fitting $SF(S_c)$, a high value for $Fit(S_c)$ does not only mean that S_c structurally fits well to the process variants, but also that a reasonable number of activities is considered in the candidate model.

Computing $Fit(S_c)$ requires only polynomial time. To be more precise, let $S_i \in \mathcal{P}$, $i = 1, 2, \ldots, n$ be a collection of process variants and let $T_i = (N_i, C_i, CT_i, E_i, l_i)$ be the corresponding process structure tree. Let further $m = |\bigcup N_i|$. The complexity for computing $Fit(S_c)$ is $O(2m^2n)$.

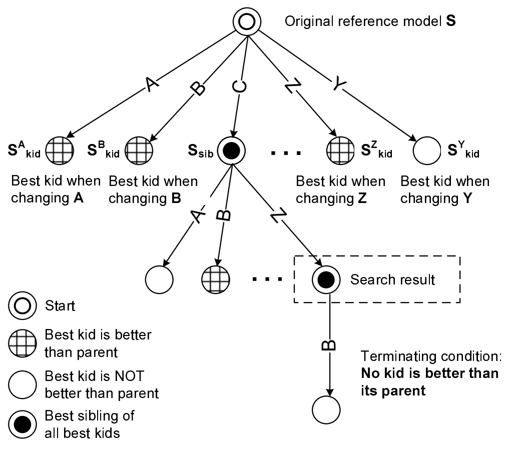
4.2. Constructing the Search Tree

We now show how to find candidate process models. We present a *best-first* algorithm for constructing a search tree to find the best candidate model in the search space.

4.2.1. The Search Tree

Remember the overview of our heuristic search approach given at the beginning of Section 4. Starting with the current candidate model S_c, in each iteration we search for its direct "neighbors" (i.e., process models with distance 1 to S_c). Thereby we try to find a better candidate model S'_c with higher fitness value. Generally, for a given process model S_c, we construct a neighbor model by applying *ONE* insert, delete, or move operation (cf. Table 1) to S_c. All activities $a_j \in \bigcup N_i$, which appear in at least one variant, are candidate activities for change. While an insert operation adds a new activity $a_j \notin N_c$ to S_c, the other two operations delete or move an activity $a_j \in N_c$ already present in S_c.

Generally, numerous process models can result when applying one change operation relating to a particular activity a_j. Note that the number of positions where we can insert activity a_j ($a_j \notin N_c$) or move it ($a_j \in N_c$) can be large. Section 4.2.2 provides details on how to find all process models that result when changing one particular activity a_j in S_c. First of all, we assume that we have already identified these neighbor models, including the one with highest fitness value (denoted as the *best kid* S^j_{kid} of S_c when changing a_j). Fig. 9 illustrates our search tree. Our search algorithm starts with setting the original reference model S as initial state, i.e., $S_c := S$ (cf. Fig. 9). We further define AS as *active activity set*, which contains all activities that might be subject to change. At the beginning $AS = \{a_j | a_j \in \bigcup_{i=1}^n N_i\}$ contains all activities that appear in at least one variant S_i. For each activity $a_j \in AS$ we then determine the corresponding best kid S^j_{kid} of S_c. If S^j_{kid} has higher fitness value than S_c, we mark it; otherwise, we remove a_j from AS (cf. Fig. 9). Afterwards, we choose the model with highest fitness value S^{j*}_{kid} among all best kids S^j_{kid}, and denote this model as *best sibling* S_{sib}. We then set S_{sib} as the first intermediate search result and replace S_c by S_{sib} for further search. Finally, we remove a_{j*} from AS.



Original reference model S
A
B
C
Z
Y
S^A kid
S^B kid
S sib
S^Z kid
S^Y kid
Best kid when changing A
Best kid when changing B
Best kid when changing Z
Best kid when changing Y
A
B
Z
Search result
B
Start
Best kid is better than parent
Best kid is NOT better than parent
Best sibling of all best kids
Terminating condition: No kid is better than its parent

Figure 9: Constructing the search tree

The described search method continues iteratively until its termination condition is met, i.e., we either cannot find a better model or the *allowed search distance* is reached. Consequently, the process engineer is flexible in controlling to what degree the discovered reference process model may differ from the original one. The final search result S_{sib} corresponds to our discovered reference model S' (the node marked by a bull's eye and circle in Fig. 9). We refer to Appendix C for an algorithm formally describing the above steps.

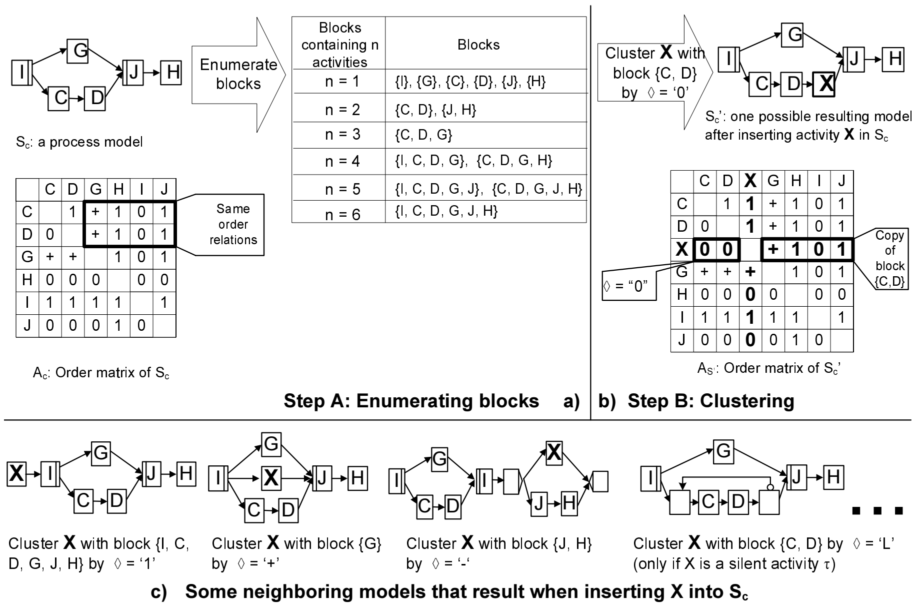
4.2.2. *Changing one Particular Activity*

Section 4.2.1 showed how to construct a search tree by comparing best kids S^j_{kid}. We now discuss how to find such best kid S^j_{kid}, i.e., how to find all "neighbors" of a candidate model S_c by performing *one* change operation relating to a particular activity a_j. Consequently, S^j_{kid} is the model with highest fitness value among all models that may results when applying one change operation on S_c relating to activity a_j. Regarding an activity a_j, we consider three types of basic change operations: *insert, delete* and *move* (cf. Table 1). The neighbor model resulting from the deletion of $a_j \in N_c$ can be easily determined by removing a_j from the process model and its order matrix [28]; movement of a_j can be simulated by deleting a_j and sub-sequently re-inserting it at the desired position. Thus, the basic challenge in finding neighbors of candidate model S_c is to apply one activity *insertion* such that the *block structuring* of the resulting model is preserved. Obviously, the positions where we can (correctly) insert a_j into S_c are our subjects of interest. Fig. 10 provides an example. Given model S_c we want to find all process models that may result when inserting `X` into S_c. We apply two steps to "simulate" this activity insertion:

Step A: Block enumeration First, we enumerate all blocks, candidate model S_c contains. A block can be an atomic activity, a sequence, a parallel branching, or S_c itself. Let $S \in \mathcal{P}$ be a block-structured process model. Let further A be the order matrix of S with activity set N. Two activities a_i and a_j can form a block iff: $\forall a_k \in N \setminus \{a_i, a_j\} : A_{a_i a_k} = A_{a_j a_k}$, i.e., they have same order relations in respect to the remaining activities. As example consider Fig. 10a: `C` and `D` can form a block since they show the same order relations concerning `G, H, I`, and `J`. As extension, two blocks B_j and B_k can be merged to a bigger one iff $[(a_\alpha, a_\beta, a_\gamma) \in B_j \times B_k \times (N \setminus B_j \bigcup B_k) : A_{a_\alpha a_\gamma} = A_{a_\beta a_\gamma}]$ holds; i.e., all activities $a_\alpha \in B_j$, $a_\beta \in B_k$ show the same order relations in respect to the activities outside the two blocks; e.g., blocks {`C,D`} and {`G`} show the same order relations in respect to activity `H,I` and `J`. Therefore they can form the bigger block {`C,D,G`}; i.e., we can determine a block containing x activities by merging two disjoint blocks containing j and k activities respectively with $x = j + k$ (cf. Fig. 10). Based on this, we are able to enumerate all blocks of different size as contained in a process model (see Appendix D for an algorithm formally describing this block enumeration).

Step B: Cluster inserted activity with one block After having enumerated all possible blocks for a given candidate model S_c, we can insert activity a_j in S_c such that we obtain a block-structured model again. Assume that we want to insert `X` in S (cf. Fig. 10). To ensure block-structuring of the resulting model, we "cluster" `X` with an enumerated block, i.e., we replace one of the previously determined blocks B by a bigger block B' containing both B and `X`. In this context, we set order relation between B and `X` either to $\diamond \in \{0, 1, +, -\}$ or $\diamond = L$ if `X` is a silent activity τ representing a loop-block; i.e., $\diamond$ defines the order relations between `X` and all activities contained in B. An example is depicted in Fig. 10. Here, the added activity `X` is clustered with block {`C,D`} using order relation $\diamond =$ "0"; i.e., `X` becomes a successor of the sequence block that contains `C` and `D`. To realize this clustering, we have to set the order relations between inserted activity `X` and block activities `C` and `D` to "0". Further, order relations between `X` and remaining activities are the same as for `C` and `D`. Finally, the three activities form a new block {`C,D,X`} replacing the old one (i.e., {`C,D`}). This way, we obtain a block-structured process model S'.

Each time we cluster an activity with a block, we actually add this activity to the position where it can form a bigger block together with the selected one; i.e., we replace a self-contained block of the process model by a bigger one. Consequently, model soundness is further guaranteed [31, 29]. Fig. 10b shows one resulting model S' we can obtain when adding `X` to S_c. Obviously, S' is not the only neighboring model since we can insert `X` at different positions; i.e., we can



Enumerate blocks
Sc: a process model
Same order relations
Ac: Order matrix of Sc
Step A: Enumerating blocks a)
Cluster X with block {C, D} by ◊ = '0'
Sc': one possible resulting model after inserting activity X in Sc
◊ = "0"
Copy of block {C,D}
AS': Order matrix of Sc'
b) Step B: Clustering
Cluster X with block {I, C, D, G, J, H} by ◊ = '1'
Cluster X with block {G} by ◊ = '+'
Cluster X with block {J, H} by ◊ = '-'
Cluster X with block {C, D} by ◊ = 'L' (only if X is a silent activity τ)
c) Some neighboring models that result when inserting X into Sc

Blocks containing n activities	Blocks
n = 1	{I}, {G}, {C}, {D}, {J}, {H}
n = 2	{C, D}, {J, H}
n = 3	{C, D, G}
n = 4	{I, C, D, G}, {C, D, G, H}
n = 5	{I, C, D, G, J}, {C, D, G, J, H}
n = 6	{I, C, D, G, J, H}

	C	D	G	H	I	J
C		1	+	1	0	1
D	0		+	1	0	1
G	+	+		1	0	1
H	0	0	0		0	0
I	1	1	1	1		1
J	0	0	0	1	0	

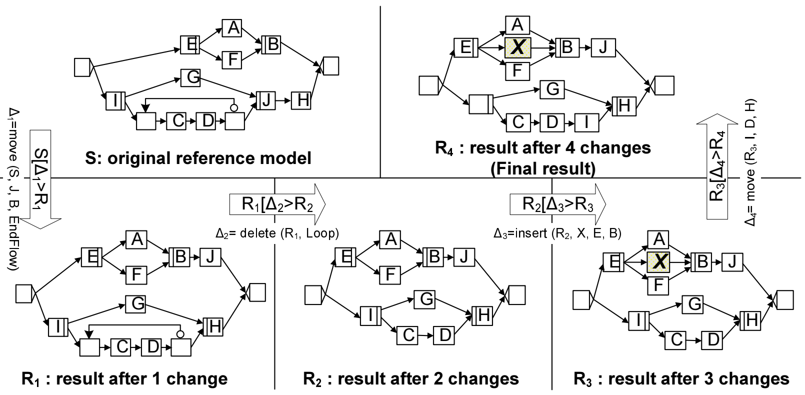
	C	D	**X**	G	H	I	J
C		1	**1**	+	1	0	1
D	0		**1**	+	1	0	1
X	**0**	**0**		**+**	**1**	**0**	**1**
G	+	+	**+**		1	0	1
H	0	0	**0**	0		0	0
I	1	1	**1**	1	1		1
J	0	0	**0**	0	1	0	

Figure 10: Finding the neighboring models by inserting X into process model S

cluster each block enumerated in Step A with X using any one of the four order relations $\diamond \in \{0, 1, +, -\}$, or by 'L' if X is a silent activity representing a loop-block. In our example from Fig. 10, S_c contains 14 blocks. Consequently, the number of models that may result when adding X in S_c corresponds to $14 \times 4 = 56$ (or $14 \times 1 = 14$ if X is a silent activity); i.e., we can obtain 56 (14) potential models. Fig. 10c shows some neighboring models of S_c. Note that the resulting models are not necessarily unique, i.e., it is possible that some of them are the same. However, this is not a critical issue since $Fit(S_c)$ can be quickly computed (cf. Section 4.1); i.e., some redundant information does not significantly decrease algorithm performance.

4.3. Search Result for our Running Example

Fig. 11 presents the search result we obtain when applying our heuristic algorithm to the example from Fig. 4. We do not set any limitation on the number of search steps in order to find the best reference model. Fig. 11 shows the evolution of the original reference model S. First operation $\Delta_1 = move(S, \text{J}, \text{B}, EndFlow)$ changes S into intermediate model R_1, which shows



S: original reference model
R4 : result after 4 changes (Final result)
S[Δ1>R1
Δ1=move (S, J, B, EndFlow)
R3[Δ4>R4
Δ4= move (R3, I, D, H)
R1[Δ2>R2
Δ2= delete (R1, Loop)
R2[Δ3>R3
Δ3=insert (R2, X, E, B)
R1 : result after 1 change
R2 : result after 2 changes
R3 : result after 3 changes

Figure 11: Search results along applied change operations

the highest fitness value in comparison to all other neighbor models of S. Using R_1 as next input for our algorithm, we discover R_2 by applying $\Delta_2 = \mathit{delete}(R_1, \mathit{Loop})$, and then R_3 using $\Delta_3 = \mathit{insert}(R_2, \texttt{X}, \texttt{E}, \texttt{B})$. Finally, we obtain R_4 by applying $\Delta_3 = \mathit{move}(R_2, \texttt{I}, \texttt{D}, \texttt{H})$ to R_3. Since we cannot find a "better" process model by changing R_4 anymore, we obtain R_4 as final result. Note that if we set constraints on allowed search steps (i.e., we only allow to change the original reference model S by maximum d change operations), the final search result will be as follows: R_d *if* $d \leq 4$ *or* R_4 *if* $d > 4$. Table 3 further compares S with all (intermediate) search results.

It is not surprising that the fitness value increases with continuing search since we use fitness to guide it. However, we need to examine whether or not the discovered process models are indeed getting better. We accomplish this by computing their average weighted distance (cf. Def. 5) to the variants, which is a precise measurement in our context. From Table 3 the iterative improvement of average weighted distance becomes clear, i.e., it drops monotonically from 4.85 to 2.4, which indicates that in the given example the algorithm performs as expected.

One design goal for our heuristic search algorithm is to be able to only consider the most important changes; i.e., the ones reducing average weighted distance between reference model and variants most, should be discovered first. We additionally evaluate *delta-fitness* and *delta-distance*, which indicate the relative improvement of fitness values and the reduction of average weighted distance after each change; e.g., operation Δ_1 first changes S into R_1, which improves fitness value (delta-fitness) by 0.143 and reduces average weighted distance (delta-distance) by 0.9. Similarly, Δ_2 reduces average weighted distance by 0.7, Δ_3 by 0.6, and Δ_4 by 0.25. Obviously, delta-distance is monotonically decreasing with increasing number of change operations. This indicates that the most important changes are performed at the beginning of the search, while less important ones are performed at the end.

Another feature of our heuristic search is its ability to automatically decide which activities shall be included in the reference model; i.e., manually filtering less relevant activities is not required. In our example, `X` is automatically added, while the loop-block is automatically deleted. The only optimization we want to achieve is to reduce average weighted distance, i.e., change operations are automatically balanced based on their contribution to reduce this measure.

4.4. Performance Evaluation based on Simulations

Using one example to measure the performance of our heuristic mining algorithm is not sufficient. Since computing the average weighted distance is at $\mathcal{NP}$*-hard* level, the suggested fitness function is only an approximation of it. Therefore, we have to analyze *to what degree delta-fitness is correlated with delta-distance.* Further, we are interested in whether important changes are performed at the beginning. If biggest distance reduction can be achieved with the first changes, setting search limitations or filtering out the change operations performed at the end, does not constitute any practical problem. Therefore, we want to know: *To what degree are important change operations positioned at the beginning of our heuristic search.*

We try to answer these questions using *simulation*; i.e., by generating thousands of data samples we provide a statistical answer [49]. We identify several parameters (e.g., size of the model,

	S	R_1	R_2	R_3	R_4
Fitness	0.543	0.687	0.805	0.844	0.859
Average weighted distance	4.85	3.95	3.25	2.65	2.4
Change Operation		Move	Delete	Insert	Move
Delta-fitness		0.143	0.118	0.039	0.009
Delta-Distance		0.9	0.7	0.6	0.25

Table 3: Search result along each applied

	Correlation analysis				Correlation comparison		
	# of activity per variant	# of data	Correlation	Signi-ficant?	Pairwise Comparison	Probability being same	Signi-ficant?
Small-sized	10 - 15	33	0.762	Yes	Small v.s. Medium	0.130	Yes
Medium-sized	20 - 30	74	0.589	Yes	Medium v.s. Large	0.689	Yes
Large-sized	50 - 75	177	0.623	Yes	Small v.s. Large	0.170	Yes

Table 4: Correlation analysis

similarity of the variants) for which we investigate whether or not they influence performance of our heuristic mining algorithm (see [50] for details); e.g., the size of the process variants ranged from 10 to 75 activities, while their similarity to the reference process model ranged from 10% to 30%. In addition, we discuss 8 different scenarios in respect to which activities and process regions are changed. Using these parameters, we generate 72 groups of datasets (7272 models in total). Each group contains a randomly generated *reference process model* and a collection of *100 different process variants*. We can generate each variant by configuring the reference model according to a particular scenario. This way we are able to evaluate the performance and robustness of our heuristic algorithm in a controlled setting. When applying heuristic mining to discover new reference models, we do not set constraints on search steps, i.e., the algorithm terminates if no better model can be discovered. *All (intermediate) process models* are documented (see Fig. 11 for an example). We compute *fitness* and *average weighted distance* for each intermediate process model. We further compute *delta-fitness* and *delta-distance* in order to examine the influence of every change operation (cf. Table 3 for an example).

Correlation of delta-fitness and delta-distance. One important issue we wanted to investigate is how delta-fitness is correlated with delta-distance. Every change operation leads to a structural modification of the process model, and consequently creates delta-fitness x_i and delta-distance y_i. In total, we perform *284* changes in our simulation when discovering reference models. We use Pearson correlation to measure correlation between delta-fitness and delta-distance [51]. Let X be delta-fitness and Y be delta-distance. We obtain n data samples (x_i, y_i), $i = 1, \ldots, n$. Let $\bar{x}$ and $\bar{y}$ be the means of X and Y, and let s_x and s_y be the standard deviations of X and Y. As Pearson correlation we then obtain $r_{xy} = \frac{\sum x_i y_i - n\bar{x}\bar{y}}{(n-1)s_x s_y}$ [51]. Results are summarized in Table 4. All correlation coefficients are *significant* and *high* (> 0.5). The high positive correlation between delta-fitness and delta-distance indicates that when finding a model with higher fitness value, we have very high chance to also reduce average weighted distance. We additionally compare these three correlations. Results indicate that they do not show significant differences to each other, i.e., they are statistically the same (see [50]). This implies that our algorithm provides search results of similar goodness *independent* from the number of activities contained in the variants.

Importance of top changes. We analyze to what degree our algorithm applies more important changes at the beginning. For this purpose, we measure to what degree the top n% changes reduce average weighted distance. As example consider search results from Table 3. We performed in total 4 change operations and reduced average weighted distance by 2.45 from 4.85 (based on S) to 2.4 (based on R_4). Among the four change operations, the first one reduces average weighted distance by 0.9. When compared to overall distance reduction of 2.45, the top 25% (i.e., the first) changes accomplish 0.9/2.45 = 36.73% of overall distance reduction. This number indicates the importance of the changes applied first. We therefore evaluate distance reduction by analyzing the top 33.3% and the top 50.0% change operations. On average, the top 33.3% change operations contribute to 63.80% distance reduction, while the top 50.0% achieve 78.93%. Consequently, changes at the beginning are *more important* than the ones performed later.

5. Scenario 2: Discovering a Reference Process Model by Mining Process Model Variants: A Clustering Approach

We now present a clustering algorithm for mining a collection of process variants without need for knowing the original reference model. Since we restrict ourselves to block-structured process models, we can build the new reference model by enlarging blocks, i.e., we first identify two activities that can form a block; then we merge this block with other activities and blocks respectively to form a larger block. This continues until all activities and blocks respectively are merged into one single block. This block and its internal structure then represent the newly discovered reference process model. Based on the aggregated order matrix (cf. Def. 10), our clustering approach for mining process variants works as follows:

Step 1.	Determine the activity set to be considered in the new reference process model.
Step 2.	Determine two activities/blocks to be clustered in a new block.
Step 3.	Determine the order relation the two clustered activities and blocks, respectively, shall have within this block.
Step 4.	After having built a new block in Steps 2 and 3, adjust the aggregated order matrix accordingly.
Step 5.	Repeat Steps 2-4 until all activities are clustered; i.e., until the new process model is constructed by enlarging blocks.

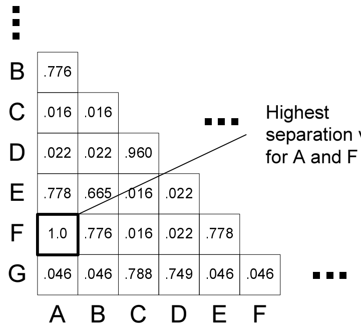
5.1. Determining the Activity Set of the Reference Process Model

One fundamental challenge is to decide which activities shall be considered in the new reference model. As basis for this decision we choose *activity frequency* (cf. Def. 12). The user may set a threshold such that only activities having an *activity frequency* higher than this threshold are considered in the reference process model. This way we can exclude activities with low frequency. As example consider Fig. 4. If we only want to consider activities with frequency greater than 60%, for instance, activities Y and Z as well as silent activity τ will be excluded from the reference process model. Excluding τ means that the loop structure is not considered. Generally, process engineers have to set a threshold depending on whether they want to add more or fewer activities to the reference model. In the following, we choose 60% as threshold value.

5.2. Determining two Activities or Blocks to be Clustered

Taking an order matrix (cf. Def. 9), two activities can form a block if they have same order relations with respect to the remaining activities (cf. Section 4.2). We can apply a similar idea to aggregated order matrices. However, for them activity relationships are expressed as 5-dimensional vector showing the distribution of the order relations over all process variants. When determining pairs of activities that can be clustered as a block, it would be too restrictive to require precise matching as in the case of an order matrix. To deal with this, we re-apply function $f(\alpha,\beta)$ (cf. Formula 6) which expresses closeness between two vectors $\alpha = (x_1, x_2, ..., x_n)$ and $\beta = (y_1, y_2, ..., y_n)$. Using $f(\alpha,\beta)$ we introduce the **Separation** metrics. It indicates to what degree two activities of an aggregated order matrix are suited for being clustered to a block. More precisely, $Separation(a, b)$ expresses how similar order relations of activities a and b are when compared to the other activities. In our example from Fig. 4, $Separation(\mathtt{A},\mathtt{B})$ is determined by the closeness (measured in terms of the cosine value) of $f(v_{\mathtt{AC}}, v_{\mathtt{BC}})$, $f(v_{\mathtt{AD}}, v_{\mathtt{BD}})$, ..., $f(v_{\mathtt{AJ}}, v_{\mathtt{BJ}})$, and $f(v_{\mathtt{AX}}, v_{\mathtt{BX}})$. We define cluster separation as follows:

$$Separation(a,b) = \frac{\sum_{x \in N \setminus \{a,b\}} f^2(v_{ax}, v_{bx})}{|N| - 2} \quad \in [0,1] \tag{8}$$


Highest separation value for A and F

	A	B	C	D	E	F
⋮						
B	.776					
C	.016	.016				
D	.022	.022	.960			
E	.778	.665	.016	.022		
F	1.0	.776	.016	.022	.778	
G	.046	.046	.788	.749	.046	.046

Figure 12: Separation table

N corresponds to the set of considered activities (cf. Section 5.1). Like most clustering algorithms [47], we square the cosine value to emphasize the differences between the two compared vectors. Finally, dividing this expression by $|N| - 2$ normalizes its value to a range between [0, 1]. The higher $Separation(a, b)$ is, the better activities a and b are separable from others, and the more probably a and b should form a block. Regarding our example from Fig. 4, we obtain $Separation(\mathtt{A},\mathtt{B}) = 0.776$. We determine the pair of activities best suited to form a block by computing the separation value for each activity pair. Fig. 12 depicts the separation values for our example from Fig. 4. We denote this table as *separation table*. Obviously, `A` and `F` have the highest separation value of 1.0. We therefore choose `A` and `F` to form our first block. Since $Separation(\mathtt{A},\mathtt{F}) = 1$ holds, `A` and `F` can form a block in all six variants (cf. Fig. 4).

5.3. Determining Internal Order Relations

After clustering `A` and `F` in the first block, we need to determine the order relation between these two activities. For this purpose, we introduce *Cohesion* to measure how significant particular order relations between two activities of the same cluster are. In the aggregated order matrix of our example, the relationship between `A` and `F` is depicted as 5-dimensional vector $v_{\mathtt{AF}} = (0, 0, 1, 0, 0)$. It shows the distribution values of the five kinds of order relations. When building a reference process model, exactly one of the five order relations is taken. Therefore, we want to choose the most significant one. Regarding our example, significance of each order relation can be evaluated by the closeness vector $v_{\mathtt{AF}}$ and the five axes in the 5-dimensional space have. These axes can be represented by five benchmarking vectors: $v^0 = (1, 0, 0, 0, 0)$, $v^1 = (0, 1, 0, 0, 0)$, $v^+ = (0, 0, 1, 0, 0)$, $v^- = (0, 0, 0, 1, 0)$, and $v^L = (0, 0, 0, 0, 1)$. We can compute the significance of each order relation using $f(\alpha,\beta)$. In our example, the closest axis to $v_{\mathtt{AF}}$ is v^+ with $f(v_{\mathtt{AF}}, v^+) = 1$. Therefore, we decide that `A` and `F` shall be organized in parallel in the newly derived block (cf. Def. 9). We use Cohesion to evaluate how good our choice is:

$$Cohesion(a,b) = \frac{\max_{\diamond \in \{0,1,+,-,L\}}\{f(v_{ab}, v^{\diamond})\} - 0.4472}{1 - 0.4472} \quad \in [0,1] \tag{9}$$

$Cohesion(a, b)$ equals *one* if there is a dominant order relation, i.e., v_{ab} is on one of the five axes. In turn, it equals *zero* if $v_{ab} = (0.2, 0.2, 0.2, 0.2, 0.2)$ holds; i.e., no order relation is more significant than the others. In our example, $Cohesion(\mathtt{A},\mathtt{F}) = 1$ holds; this indicates that `A` and `F` have order relation '+' in all six process variants; we obtain the same results when directly analyzing the variants (cf. Fig. 4).

5.4. Recomputing the Aggregated Order Matrix

We discovered the first block of our reference model which contains `A` and `F` with order relation '+'. We now have to set the relationship between newly created block and remain-

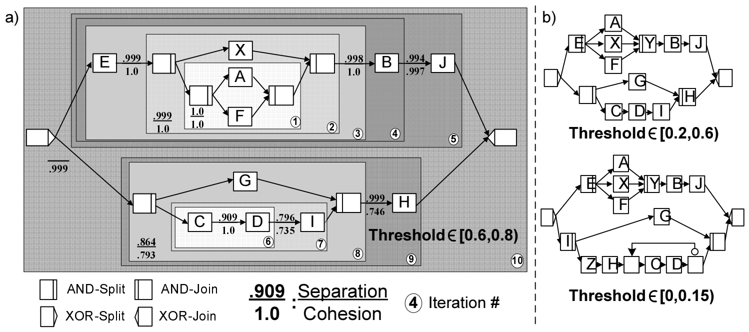


Figure 13: Reference process model discovered by clustering algorithm

ing activities. We accomplish this by adapting the aggregated order matrix.[10] For this, we compute the means of the order relations between {`A`, `F`} and remaining activities; e.g., since $v_{\mathtt{AI}} = (0, 0.15, 0, 0.85, 0)$ and $v_{\mathtt{FI}} = (0, 0.15, 0, 0.85, 0)$ hold, the order relation between the new block {`A`,`F`} and activity `I` corresponds to $(v_{\mathtt{AI}} + v_{\mathtt{FI}})/2 = (0, 0.15, 0, 0.85, 0)$.[11] Such computation is applied to all remaining activities outside this block. Generally, after clustering two activities `a` and `b`, the aggregated order matrix V' is re-calculated as follows:

$$\forall x \in N \setminus \{a, b\} : \begin{cases} v'_{(a,b)x} = (v_{ax} + v_{bx})/2 \\ v'_{x(a,b)} = (v_{xa} + v_{xb})/2 \\ v'_{xy} = v_{xy}, v'_{yx} = v_{yx} \quad \text{for} \quad \forall y \in N \setminus \{a, b, x\} \end{cases} \tag{10}$$

Since `A` and `F` are replaced by one block, the matrix resulting from this re-computation is one dimension smaller than V. Afterwards, we treat this block like a single activity, but *keep its internal structure* in order to build up the new reference process model at the end.

5.5. Applying the Clustering Algorithm to our Example

We re-apply the steps described in Sections 5.2 - 5.4 until all activities and blocks respectively are clustered together. Fig. 13a shows the reference process model we can discover for our example. It further depicts the blocks as constructed in each iteration as well as the two evaluation measures *Separation* and *Cohesion*. Using separation and cohesion, we can evaluate how each part of the reference process model fits to the variants. For example, it is clear that activities `A` and `F` can always form a block in the six variants (high separation) and the order relation between `A` and `F` is also consistent (high cohesion). By contrast, activity `I` does not often succeed block {`C`,`D`} (low separation and cohesion). We can draw similar results if we have another look at the process variants (cf. Fig. 4). Fig. 13b further shows two other reference process models we obtain when setting different threshold values for determining the activity set (cf. Section 5.1).

[10]Our approach is different from traditional clustering algorithms [47], which only re-compute distances, but not the original dataset.

[11]This approach is an unweighted one; i.e., we simply take the average of the two vectors without considering their importance (e.g., how many activities are included in the block). This way, when merging two blocks of different size, we can ensure that the order relations of the resulting block are not too much dominated by the bigger one. Such unweighted approach is widely used in existing clustering algorithms [47].

5.6. Proof-of-Concept Prototype

We implemented and tested the heuristic and clustering algorithm using Java. Fig. 14 depicts a screenshot of our prototype. We use the ADEPT2 Process Template Editor [30] as tool for creating process variants. For each process model, the editor can generate an XML representation with all relevant information being marked up. We store variants in a repository which can be accessed by our mining procedure. The mining algorithms were developed as stand-alone Java program, independent from the process editor. This program can read the original reference model (if available) as well as all process variants. It then generates the result models and stores them as accessible XML schemas. All intermediate search results are also stored.

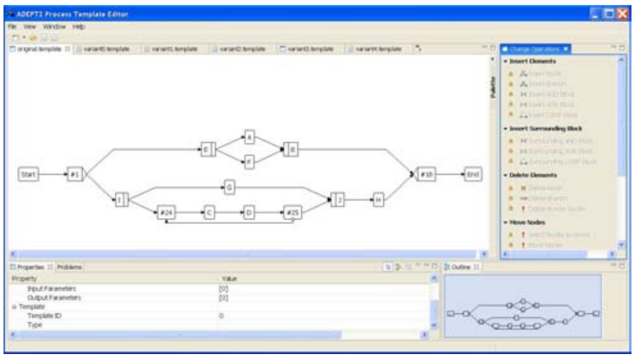


Figure 14: Screenshot of the prototype

6. Algorithm Comparisons

Sections 6.1 and 6.2 compares our heuristic algorithm with our clustering approach. Section 6.3 then compares the two algorithms with existing process mining techniques [18], i.e., algorithms that discover process models from execution logs.

6.1. Qualitative Comparison of the Algorithms for Process Variant Mining

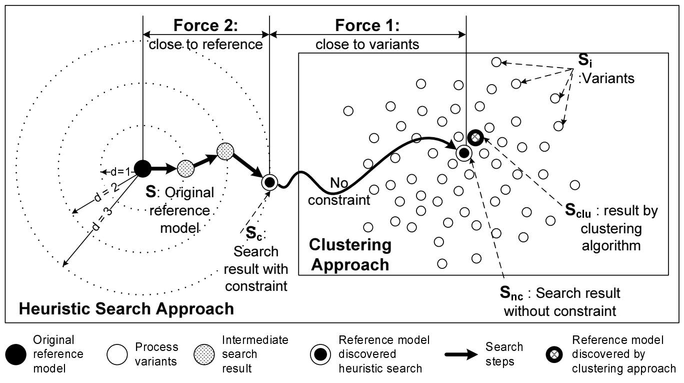

Force 2:
close to reference
Force 1:
close to variants
S_i :Variants
d=1
d=2
d=3
S: Original reference model
S_c: Search result with constraint
No constraint
Clustering Approach
S_{clu} : result by clustering algorithm
S_{nc} : Search result without constraint
Heuristic Search Approach
Original reference model
Process variants
Intermediate search result
Reference model discovered heuristic search
Search steps
Reference model discovered by clustering approach

Figure 15: High-Level overview of the two algorithms

Inputs and Goals. Fig. 15 illustrates how our heuristic and clustering mining algorithms differ in respect to goals and inputs. It represents each process variant S_i as single (white) node in the 2-dimensional space. Our heuristic algorithm tries to discover a new reference process model by applying a sequence of change operations to the original one. In particular, it balances two "forces": one is to bring the new reference model S_c closer to the variants (i.e., to the bull's eye S_{nc} at the right), the other force is to not "move" it too far away from the original reference model S; i.e., S_c should not differ too much from S. Our heuristic algorithm provides such flexibility by allowing process engineers to set a maximum search distance. Our simulations (cf. Section 4.4) showed that the change operations which are applied first to the (original) reference model are more important than the ones positioned at the end i.e., they reduce distance between reference model and variants to a larger extent. Consequently, when ignoring less relevant changes we do not influence overall distance reduction too much.

While the above scenario presumes knowledge of the original reference model, we should be able to cope with cases in which there exists only a variant collection, but the original reference process model is unknown. In this scenario, the goal of our clustering approach is to discover the "center" of the variants, i.e., a reference process model with shortest average weighted distance to them. In principle, we can also apply our heuristic algorithm to this scenario. We just need to start with an "empty" model S without setting any search limitation. However, since we do not need to balance the two forces and to perform the important change operations at the beginning of the search, the clustering algorithm is expected to be faster and to provide additional information on the search result (see Section 6.2).

Design Principles and Complexity. Our heuristic algorithm discovers a better reference model by applying a sequence of change operations to the original one. To enable quick decisions for a large search space (cf. Section 4.2), we use a fitness function to evaluate how well a candidate model fits to the variants. This fitness function only provides a global evaluation, but does not show how each part of the candidate model fits to the variants. On the contrary, the clustering algorithm discovers a reference process model by enlarging blocks. By evaluating separation and cohesion, we are able to determine how well each part of the discovered reference model fits to the variants; i.e., due to its different design the clustering algorithm returns more information than the heuristic one. Complexity of the two algorithms differs as well. Despite polynomial complexity for computing the fitness of a candidate model, enumerating all blocks in a candidate model has $\mathcal{NP}$*-hard* complexity.[12] On the contrary, our clustering algorithm has polynomial complexity since computing separation and cohesion are both polynomial. To be more precise, if m is the number of activities and n the one of variants, complexity of the clustering algorithm corresponds to $O(m^2n + m^3)$. This implies that the clustering algorithm can quickly compute the reference process model even for a large variant collection, while the heuristic algorithm may take considerable longer.

Pros & Cons. Table 5 summarizes the differences between the two algorithms. Additional attention should be paid to their pros & cons. Since the clustering algorithm has polynomial complexity, it runs significantly faster than the heuristic one. Using Separation and Cohesion we obtain information on how each part of the discovered reference process model fits to the variants. However, our clustering algorithm cannot control the discovery procedure or distinguish important changes from less relevant ones as our heuristic algorithm does. Though for our running example the clustering algorithm discovered the same process model (cf. Fig. 13) as our heuristic algorithm (cf. Fig. 11), in many other cases the model discovered by the clustering algorithm was not as good as the one discovered by the heuristic algorithm (cf. Section 6.2 for details). Reason is that the heuristic algorithm searches a significantly larger space which contains a large number of candidate process models.

6.2. Quantitative Comparison of the Algorithms for Process Variant Mining

We now compare the two algorithms quantitatively by analyzing how fast they run and how good the discovered models are. We use the same data for this comparison as for the evaluation of

[12]Worst-case, complexity of this algorithm is 2^n where n corresponds to the number of activities. This worst-case scenario will only occur if any combination of activities may form a block (like a process model for which all activities are ordered in parallel to each other). During our simulation, in most cases we were able to enumerate all blocks of a process model within milliseconds. This indicates rather good performance in practice.

	Clustering Algorithm	**Heuristic Algorithm**
Input	Collection of process variants.	Collection of process variants + original reference process model
Goal	Discover reference process model with shortest average weighted distance to the variants	Discover better reference process model with maximum distance to the original one
Use cases	Scenario 1 (cf. Section 1.1).	Scenario 2 (cf. Section 1.1).
Design principle	Local view: Discover reference process model by enlarging blocks	Global view: Discover reference process model by searching for better candidate models
Complexity	$O(m^2n + m^3)$ (m : # activities; n : # variants)	Sub-steps contain $\mathcal{NP}$-*hard* problems
Pros & Cons	1. Runs very fast 2. Provides local view on how each part of the reference process model fits to the variants 3. Activity set can be flexibly chosen by user	1. Automatically selects the activity set 2. Can control the maximum distance between the original reference process model and the discovered one 3. Applies more important changes at the beginning

Table 5: Qualitative comparison between clustering algorithm and heuristic algorithm

our heuristic algorithm (cf Section 4.4). We generate 72 groups of datasets representing different scenarios. Each group contains 1 reference process model and 100 process variants. Based on this, by applying each of the two algorithms we discover a new reference process model and document the relating execution time and distance reduction (between discovered model and original one). Results are summarized in Table 6. They indicate that the clustering algorithm runs significantly faster than the heuristic one. However, results obtained with the clustering algorithm are not as good as the ones provided by the heuristic algorithm.

	Average execution time			Average distance reduction	
	# activities per variant	Clustering Algorithm	Heuristic Algorithm	Clustering Algorithm	Heuristic Algorithm
Small-sized	10 - 15	0.013	0.184	6.93%	19.73%
Medium-sized	20 - 30	0.022	4.568	11.14%	22.59%
Large-sized	50 - 75	0.181	805.539	-8.97%	11.70%

Table 6: Comparing the performance of the clustering and heuristic algorithms

6.3. Comparison with Existing Process Mining Algorithms

Process mining has been extensively studied in literature [18, 19, 17, 20]. Its key idea is to discover a process model by analyzing the *execution behavior* of process instances as captured in execution logs [18]. The latter document the start/end of each activity execution. Form this we can obtain a set of traces (cf. Def. 2), which reflect the behavior of implemented processes. In principle, process mining can be applied in our context as well. Consider our example from Fig. 10. For each process variant S_i we could first obtain its trace set $\mathcal{T}_{S_i}$ by enumerating all traces producible by S_i [52]. If a process model contains loop structures (i.e., it can generate infinite number of traces), without loss of generality, we assume that a loop-block is executed either once or twice. Despite this simplification, the number of traces producible by a process model can be extremely large; e.g., if a parallel branching contains five branches, of which each contains five activities, the number of producible traces corresponds to $(5 \times 5)!/(5!)^5 = 623360743125120$. This explains why we do the comparison only in small scale.[13]

[13]Note that the main goal of process mining algorithms is to discover a process model based on the traces captured in the execution log. In most cases, an execution log only captures a small fraction of the traces producible by the underlying process model [18, 17, 20, 19]. This means that process mining does not really require enumerating all traces producible by a process model for further analysis. In the context of our algorithm comparison, we decide to enumerate

The trace sets (cf. Def. 2) generated for the variants are merged into one trace set $\mathcal{T}$ taking the weight of each variant into account. For example, as S_1 accounts for 25% of the variants, we ensure that each trace producible by S_1 has the same number of instances and that the sum of all instances producible by S_1 accounts for 25% of the instances in $\mathcal{T}$ as well. We consider $\mathcal{T}$ as execution log since it fully covers the behavior of the given variant collection.

Since all activities contained in execution logs will be included in the process model discovered by process mining algorithms (like in our clustering algorithm), we introduce two additional datasets. The first one filters out all activities a_j whose activity frequency $g(a_j)$ is lower than 0.2 regarding the given variant collection (cf. Def. 12); i.e., in our example activity Z in S_5 and silent activity τ (representing the loop in S_4) are ignored. For this extended data set, we determine trace set $\mathcal{T}_{0.2}$. In the second dataset, we filter out the activities with an activity frequency lower than 0.6. Regarding our example, besides Z and τ we filter out activity Y in S_2, S_3, S_5, and S_6. Consequently, we obtain trace set $\mathcal{T}_{0.6}$ which contains all traces producible by the reduced variants. Note that $\mathcal{T}_{0.6}$ has the same activity set as the model discovered by our heuristic algorithm (cf. R_4 in Fig. 11). The enumerated trace sets $\mathcal{T}$, $\mathcal{T}_{0.2}$ and $\mathcal{T}_{0.6}$ are imported into the ProM framework, one of the most powerful tool for process mining and analysis [53]. In our comparison, we consider alpha algorithm [18], heuristic miner [19], genetic mining [17], and multi-phase miner [20]. These are well-known algorithms for discovering process models from execution logs.[14]

6.3.1. Evaluation Criteria

Our algorithms focus on the *structural* perspective of process models, i.e., our goal is to configure the variant models out of a reference model with minimal efforts (i.e., requiring a minimum number of high-level changes). On the contrary, traditional process mining focuses on process *behavior*, i.e., the discovered process model should cover the behavior of the variant models (as reflected by their trace sets) [18, 17, 20, 19]. In the following, we compare our algorithms with existing process mining algorithms from both a structural and a behavioral perspective.

Since most existing process mining algorithms discover Petri Nets or Event Process Chains (EPCs), we transform the process models discovered by the different algorithms into respective representations in order to enable their comparison (see [54, 52] for transformation techniques). Particularly, such model transformation enables us to reuse existing metrics [16, 24] and tools [53] for evaluating process models. We briefly describe the metrices we apply and refer to [16, 24, 53] for details. We first introduce three parameters to evaluate the structure of process models, namely *average weighted distance, structural appropriateness*, and *# splits/joins in EPC*.

1. **Average weighted distance** measures the efforts to configure the process variants out of

all traces producible by each variant for the following two reasons:

1. In the context of our research, we do not assume the existence of an execution log. However, most process mining evaluation criteria rely on traces, e.g., *fitness, successful execution*, *proper completion*, and *behavioral appropriateness* (cf. Section 6.3.1). Therefore, we adopt a general approach to first enumerate all traces and then to discover a process model. This way, we are able to compare all related mining algorithms based on the same approach.

2. Alternatively, we can randomly enumerate a collection of traces producible by a process model, and consider these random traces as execution log partially reflecting the behavior of the process model. However, we are not aware of any technique which can enumerate a representative set of traces (but not all traces) to express the behavior of a process model. If we randomly select a fraction of traces, we cannot ensure a fair comparison since the results significantly depend on the randomly selected trace set. Therefore, we decide to enumerate all traces producible by a process model. This way, we ensure that all possible behavior is considered, and results are not influenced by randomness.

[14]The enumerated trace sets $\mathcal{T}$, $\mathcal{T}_{0.2}$ and $\mathcal{T}_{0.6}$ as well as the process models discovered by the different algorithms are available at *http://wwwhome.cs.utwente.nl/_lic/Resources.html*.

the discovered reference process model; the lower it is the easier the variants can be configured.

2. **Structural appropriateness** measures the complexity of a Petri Net by computing the ratio between labeled transitions and nodes (transitions and places) [16]. The value range of this parameter is [0,1]; the higher it is, the simpler the Petri Net is.
3. **# Splits/Joins in EPC** measures the number of splits / joins of an EPC, and thus measures its complexity [24]. The higher it is, the more choices end users need to make when executing the EPC.

We additionally use three parameters to evaluate the behavior of the discovered process models, namely *behavior fitness, successful execution* and *proper completion*.

1. **Behavior fitness** evaluates whether the discovered process model (represented as Petri Net) complies with the behavior as captured in the trace set [16]. One way to investigate behavior fitness is to replay the traces on the Petri net. This is done in a non-blocking way, i.e., if there are missing tokens to fire a transition in the discovered model, they are artificially created and replay proceeds [16]. The value range of this parameter is [0,1]. The higher behavior fitness is, the better the trace set will be covered by the model.
2. **Successful execution** measures the percentage of traces in the trace set that can be successfully executed by the discovered process model [16]. Its value range is [0,1]. The higher it is, the more traces can be re-produced based on the discovered model.
3. **Proper completion** measures the percentage of those traces that lead to proper completion [16]. When compared to ”successful execution” this parameter requires that the analyzed process model reaches an end state when replaying a trace. The value range of this parameter is [0,1]. The higher it is, the more traces lead to proper completion.

		Structure measurement			Behavior measurement		
Dataset	Algorithms	Average weighted distance	Structural appropriateness	# Joins/ splits in EPC	Behavior fitness	Successful execution	Proper completion
$\mathcal{T}$	Heuristic Var.	2.4	0.481	6	0.876	0.353	0.353
	Clustering	4.75	0.468	8	0.737	0.120	0.120
	Alpha	8.55	0.441	15	0.646	0	0
	Heuristic	8.85	0.258	31	0.437	0.042	0
	Genetic	6.6	0.341	19	0.811	0.342	0.009
	Multi-phase	2245 arcs and 515 transitions		19	In theory, all equals 1		
$\mathcal{T}_{0.2}$	Heuristic Var.	2.4	0.481	6	0.886	0.382	0.382
	Clustering	2.6	0.482	6	0.784	0.133	0.133
	Alpha	6.9	0.466	12	0.706	0	0
	Heuristic	8.2	0.274	12	0.789	0.268	0
	Genetic	5.9	0.424	13	0.846	0.460	0.009
	Multi-phase	1534 arcs and 384 transitions		18	In theory, all equals 1		
$\mathcal{T}_{0.6}$	Heuristic Var.	2.4	0.481	6	0.851	0.327	0.337
	Clustering						
	Alpha	6.85	0.5	7	0.814	0.407	0
	Heuristic	7.85	0.462	10	0.736	0.407	0
	Genetic	3.2	0.325	13	0.886	0.394	0.278
	Multi-phase	1266 arcs and 302 transitions		17	In theory, all equals 1		

Table 7: Performance comparison with process mining algorithms

6.3.2. Evaluation Results

Evaluation results are summarized in Table 7. To differentiate our heuristic algorithm from heuristic miner known from process mining [19], we denote it as "Heuristic var." in Table 7. Without surprise, our heuristic and clustering algorithms discover a process model of simple structure. Independent from the chosen dataset, the process models discovered by these algorithms have better scores for parameters relating to the structure of the discovered model; i.e., they show lower average weighted distance, higher structural appropriateness, and lower number of splits/joins in the corresponding EPC. Except multi-phase miner [20], none of the algorithms discovered a process model with behavior fitness being 1. Note that multi-phase miner was designed in a way that it always discovers a process model with fitness 1. Despite the fact that the models discovered by multi-phase miner are very complex, they allow for more behavior not covered by the variants [20, 16]; i.e., results are often overfitting.[15] Consequently, we consider the costs of multi-phase miner for reaching a behavior fitness of 1 as too high. When excluding multi-phase miner, evaluation results show that even if we apply traditional process mining algorithms for discovering a process model that covers the behavior of the variants best, the resulting model might NOT be able to support all behavior captured by the variants. This indicates the necessity for process configuration: i.e., it is not sufficient to maintain only one model covering all behavior. Instead we must enable process configurations at both run- and build-time to obtain different process variants supporting specific behaviors in different scenarios.

For the given dataset, behavior measurements of the process model discovered by our heuristic algorithm are good as well. Note that our heuristic algorithm discovered the same model (cf. R_4 in Fig. 11) for $\mathcal{T}$, $\mathcal{T}_{0.2}$ and $\mathcal{T}_{0.6}$. This model has highest behavior fitness for trace sets $\mathcal{T}$ and $\mathcal{T}_{0.2}$, and only a few percent less than the genetic algorithm for $\mathcal{T}_{0.6}$. This was unexpected because our heuristic algorithm is focusing on structure rather than on behavior. Though our algorithm focuses on the discovery of a reference model out of which the process variants can be easily configured, this implies that behavior of the discovered model has not been sacrificed that much. Since behavior fitness is not 1, however, we should apply process configurations to obtain suited process variants supporting the execution of different process instances best.

7. Applying the Algorithms to a Practical Case

We applied our algorithms in a cast study in order to evaluate their practical benefits.

Context. We conducted the case study in a large automotive company in which we analyzed the variants of its product change management process. Basically, this process comprises several major phases like specification of a change request, handling of this change request, change implementation, and roll-out. In the following we only consider the top-level process and comment on sub-processes later on. Usually, the change management process starts with the initiation of a Change Request (CR), which must then be detailed and assessed by different teams (e.g., from engineering and production planning). The gathered comments then have to be aggregated and approved by the CR board. In case of positive approval, change implementation starts, e.g., detailing the planning and triggering the re-engineering of parts affected by the change.

[15]In principle, it is possible to measure overfitting using *behavioral appropriateness* [16]. However, due to the complexity of the discovered models, the conformance checker of ProM cannot measure some of the models (besides Multi-phase miner) in a reasonable time (e.g., within a couple of days). Therefore we did not include it in our comparison.

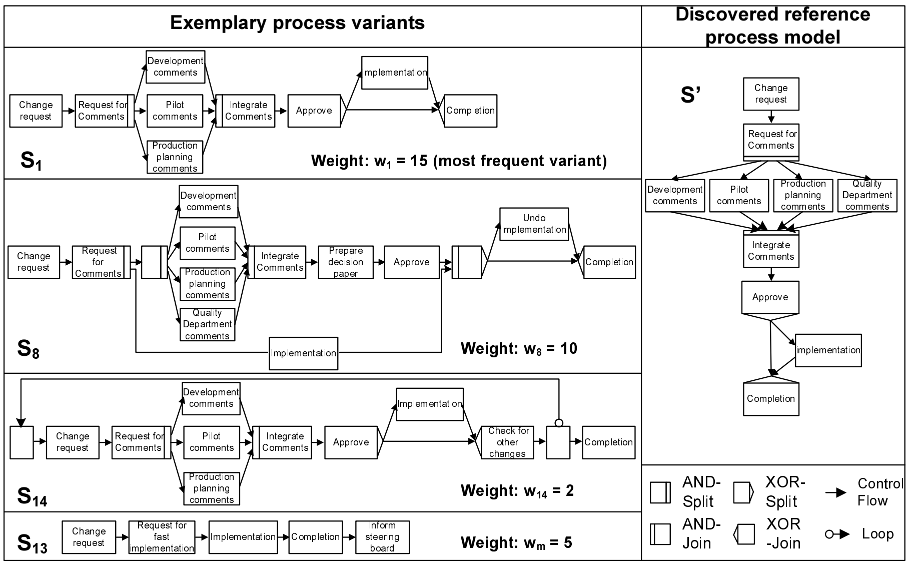


Figure 16: Example process variants in the change management case study

Data Source. We identified 14 process variants dealing with (product) change management. These variants were captured in separate process models being expressed in terms of UML Activity Diagrams and using standard process patterns like Sequence, AND-/XOR-Splits, AND-/XOR-Joins, and Loop. The size of the considered variants ranged from 5 to 12 activities and their weights, which express the number of instances running on these variant models, ranged from 2 to 15. However, none of the process variants was dominant or significantly more relevant than others. All variant models either were already block-structured or could be transformed into a behavior-equivalent block-structured process model.

Sources of Variance. Though the variant models show structural similarities they comprise parts which are only relevant for a sub-collection of the variants. For critical changes, for example, the Quality Assurance Department needs to be involved in the appraisal and commenting of the change request, while this is not required for normal changes. Concerning low-cost changes, in turn, change implementation may start before the change request is approved. In this case, the implementation procedure will have to be aborted and compensated if the approval is withhold. Other points of variation concern the preparation of the approval task, the communication of implemented changes, and the triggering of secondary changes (raised by the requested one). The left of Fig. 16 exemplarily shows 4 variant models from this case.

Case Study Results. Since we did not know the original reference process model, this case corresponds to Scenario 2. We first applied our clustering algorithm to "merge" the process variants. This way we obtained S' (cf. Fig. 16) as reference process model. As average weighted distance between S' and the variants we obtained 2.06. The time to find the model was negligible (0.031 seconds). We further applied our heuristic algorithm to the given case. Since there was no original reference process model, we used the most frequent variant (cf. S_1 in Fig. 16) as starting point of our search. We did not set any search limitation in order to discover the best model. Again we obtained S' (cf. Fig. 16) as best reference process model (after performing one change on S_1). Though our heuristic algorithm ran longer than the clustering one to find the reference process model, overall search time was only 1.062 seconds. We discussed the discovered reference model with process engineers from the company who confirmed that it constitutes a good choice for the top level change management process.

Based on the discovered reference process model, we can apply advanced techniques to configure it into the different process variants in an effective and manageable way. In the automotive company, in which we conducted the case study, the Provop research project was launched in which advanced concepts for managing and configuring process variants have been developed [10]. In Provop a particular process variant can be configured out of a given reference process model by domain experts by applying a set of high-level, pre-specified changes.

The presented mining algorithms can significantly speed up the design of such reference process model. More precisely, respective reference models can be automatically discovered for any collection of block-structured process variants. When further applying our mining algorithms to sub-processes relating to the different phases of the change management process (e.g., change implementation) and their variants we obtained good results as well.

Discussion. The case constitutes one of many process scenarios we encountered and analyzed in the automotive domain. Interestingly, for almost all of them we were able to identify large collections of similar process variants, each of them being valid in a particular application context. Regarding the presented case, process owners liked the discovered reference process model and considered it as being intuitive. Based on this result, they asked us to apply our mining approach to the more specific phases of the change management process as well, which resulted in well accepted reference models for its sub-processes as well. We also studied other sources of data. Regarding release management for electric/ electronic components in a car, for example, we identified more than 20 process variants depending on the product series, involved suppliers, or considered development phases. Another complex scenario we considered was the product creation process, for which dozens of variants exist. Thereby, each variant is assigned to a particular product type (e.g., car, truck, or bus) with different organizational responsibilities and strategic goals, or varying in some other aspects. Regarding the latter case, however, we encountered additional problems concerning the inconsistent labeling of activities, the use of different process granularities, and the heterogeneity of the used modeling formalisms. This also shows that our algorithms need to be integrated in a larger process repository framework, which additionally provides support for model configuration, refactoring, and management [15, 10].

Case studies in other domains. The practical benefit of our algorithms became evident in the context of another case study we conducted in a clinical centre. We analyzed more than 90 process variants for handling medical orders (e.g., X-ray inspections, lab tests). By applying our algorithms to these 90 variants we obtained a reference model that was significantly closer to the variants than the old reference model. The discovered reference process model was adopted by the clinical centre [55].

8. Related Work

Though algorithms applying heuristic search and clustering techniques have been widely used in data mining [47], artificial intelligence [46], and machine learning [48], only few approaches apply heuristics or clustering techniques in the context of process variant management. In particular, only few solutions exist for learning from the adaptations that were applied when configuring a collection of process variants out of a reference process model.

Structural process changes at runtime and approaches for flexible process configuration have been intensively discussed [11, 4]. A comprehensive analysis of theoretical and practical issues related to (dynamic) process changes has been provided in the ADEPT2 change framework [8].

Further, there exist approaches for dynamically changing the structure of Petri nets [40]. Based on such frameworks, the AristaFlow BPM suite [30] and tools for configurable process models [56] emerged. [57, 58] additionally present repository services for storing, managing, and querying large collections of process variants. Graph-based search techniques are used for retrieving variants that are similar to a process fragment specified by the user. Obviously, this requires profound knowledge about the structure of stored processes. Apart from this, no techniques for analyzing the different variants and for learning from their specific customizations are provided.

ProCycle enables change reuse at the process instance level to effectively deal with recurrent problem situations [11]. ProCycle applies case-based reasoning techniques to allow for the semantic annotation as well as the retrieval of process changes. Respective process adaptations can be reused in similar problem context later when configuring other process instances. If the reuse of a particular change exceeds a certain threshold, it becomes a candidate for adapting the process model at type level. Though the basic goal of ProCycle is similar to our approach, its techniques are much simpler and do not consider change variation.

A process model can be represented as graph structure which enables different kinds of graph-based analyses [59, 47, 60, 61]. Informally, a graph consists of set of nodes, which can be connected using (directed) edges. Regarding graph representations there exist only few techniques which foster learning from process variants by mining recorded change primitives (e.g., to add or delete control edges). [59] measures process model similarity based on change primitives and suggests mining techniques using this measure. Similar techniques exist in the field of association rule mining [47], frequent sub-graph mining [60] and graph pattern discovery [61]; here common edges between different nodes are discovered to construct a common sub-graph from a set of graphs. We refer to [62] for a survey on graph mining topics and algorithms. However, graph-based approaches do not consider important features of a process meta model; e.g., they are unable to deal with silent activities, cannot differentiate between AND- and XOR-branchings or Loops, and cannot guarantee the soundness of the discovered process model.

Considering Configurable Workflow Models [56], all process variants are merged into one reference model based on inheritance rules known from Petri Nets [40]. Though techniques like questionnaire-based configuration contribute to make the right configuration decisions [63], the resulting model turns out to be complex and often contains numerous decision points (see the case reported in [64]). This approach even becomes more difficult when dealing with a large collection of process variants not being equally important. In this case, an extremely large or complex process model results which contains too many decision points and cannot differentiate between important variants and trivial ones. In fields like healthcare, such complex models are not preferred due to the resulting configuration efforts [65, 2].

To mine high-level change operations, [14] presents an approach based on process mining techniques, i.e., the input consists of a change log and process mining algorithms are applied to discover the execution sequences of the changes (i.e., the change meta process). However, this approach simply considers each change as individual operation such that the result is more like a visualization of changes rather than their mining. [66] introduces a technique to rank activities based on their potential involvement in process configurations. However, it cannot provide suggestions on how to involve these activities in model changes to improve the reference process model.

9. Summary and Outlook

We presented challenges, scenarios and algorithms for mining a collection of process variants. In particular, we introduced, evaluated and compared two algorithms for discovering a reference process model out of a collection of block-structured process variants. Adopting the discovered model as new reference process model makes future process configuration easier, since less efforts for configuring the variants will be required. Our heuristic algorithm can take the original reference model into account such that users can control to what degree the discovered model differs from the original one. This way, we can avoid Spaghetti-like process models, and also control how many changes we want to perform. Through a simulation of several thousands process models we learned that the heuristic algorithm applies the important changes at the beginning of the search and is able to scale up. The clustering algorithm, in turn, does not presume any knowledge about the original reference process model the process variants were configured from. By only looking at the variant collection, it can quickly discover a reference process model in polynomial time and provide additional information on how well each part of the discovered reference model fits to the variants. We successfully applied the two algorithms to practical cases. We further compared them with existing process mining algorithms. Results indicate good performance of our algorithms in both structure and behavior aspect. However, it would be useful to integrate them with existing process mining algorithms such that we can take both the structural and the behavioral perspective into account in order to cover more general cases [18]. As we learned, data-flow also constitutes an important part of process configurations. Therefore, we want to additionally consider this perspective in future research.

Acknowledgment: This work was done in the MinAdept project, which has been supported by the Netherlands Organization for Scientific Research under contract number 612.066.512.

References

[1] B. Mutschler, M. Reichert, and J. Bumiller. Unleashing the effectiveness of process-oriented information systems: Problem analysis, critical success factors and implications. *IEEE Trans. Sys. Man. & Cyb.*, 38(3):280–291, 2008.

[2] R. Lenz and M. Reichert. IT support for healthcare processes - premises, challenges, perspectives. *Data & Knowledge Engineering*, 61(1):39–58, 2007.

[3] T.H. Davenport. *Mission Critical - Realizing the Promise of Enterprise Systems.* Harvard Business School, 2000.

[4] B. Weber, M. Reichert, and S. Rinderle-Ma. Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data and Knowledge Engineering*, 66(3):438–466, 2008.

[5] B. Weber, S. Sadiq, and M. Reichert. Beyond rigidity - dynamic process lifecycle support: A survey on dynamic changes in process-aware information systems. *Computer Science - R&D*, 23(2):47–65, 2009.

[6] A. Hallerbach, T. Bauer, and M. Reichert. Managing process variants in the process lifecycle. In *ICEIS '08*, pages 154–161. Springer, 2008.

[7] M. Rosemann and W.M.P. van der Aalst. A configurable reference modelling language. *Information Systems*, 32(1):1–23, 2007.

[8] M. Reichert and P. Dadam. ADEPTflex - supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

[9] M. Rosenmann. Potential pitfalls of process modeling: part B. *BPM Journal*, 12(3):127–136, 2006.

[10] A. Hallerbach, T. Bauer, and M. Reichert. Capturing variability in business process models: the Provop approach. *Journal of Software Maintenance and Evolution*, 22(6-7):519–546, 2010.

[11] B. Weber, M. Reichert, W. Wild, and S. Rinderle-Ma. Providing integrated life cycle support in process-aware information systems. *Int'l Journal of Cooperative Information Systems*, 19(1):115–165, 2009.

[12] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In *CAiSE'00*, pages 431–445. LNCS 1789, Springer, 2000.

[13] R.M. Dijkman, M. Dumas, L. Garcia-Banuelos, and R. Kaarik. Aligning business process models. In *EDOC'09*, pages 45–53, 2009.

[14] C.W. Günther, S. Rinderle-Ma, M. Reichert, W.M.P. van der Aalst, and J. Recker. Using process mining to learn from process changes in evolutionary systems. *Int'l J. of Business Process Int. and Mgmt*, 3(1):61–78, 2008.

[15] B. Weber, M. Reichert, J. Mendling, and H.A. Reijers. Refactoring large process model repositories. *Computers in Industry*, page accepted for publication.

[16] A. Rozinat and W.M.P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1):64–95, 2008.

[17] A.K. Alves de Medeiros. *Genetic Process Mining*. PhD thesis, Eindhoven University of Technology, NL, 2006.

[18] W.M.P van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1128–1142, 2004.

[19] A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering workflow models from event-based data using little thumb. *Integr. Comput.-Aided Eng.*, 10(2):151–162, 2003.

[20] B.F. van Dongen and W.M.P. van der Aalst. Multi-phase process mining: Building instance graphs. In *ER'04*, pages 362–376. LNCS 3288, Springer, 2004.

[21] C. Li, M. Reichert, and A. Wombacher. Discovering reference models by mining process variants using a heuristic approach. In *BPM'09, LNCS 5701*, pages 344–362. Springer, 2009.

[22] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[23] M. zur Muehlen and J. Recker. How much language is enough? Theoretical and practical use of the business process modeling notation. In *CAiSE'08*, pages 465–479. LNCS 5074, Springer, 2008.

[24] J. Mendling. *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction and Guidelines for Correctness*, volume 6 of *LNBIP*. Springer, 2008.

[25] BPEL. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[26] W.M.P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[27] J. Mendling, B.F. van Dongen, and W.M.P. van der Aalst. Getting rid of OR-joins and multiple start events in business process models. *Enterprise Information Systems*, 2(4):403–419, 2008.

[28] C. Li, M. Reichert, and A. Wombacher. On measuring process model similarity based on high-level change operations. In *ER '08*, pages 248–262. Springer LNCS 5231, 2008.

[29] M. Reichert. *Dynamische Ablaufänderungen in Workflow-Management-Systemen*. PhD thesis, Ulm University, Germany, 2000.

[30] M. Reichert, S. Rinderle, U. Kreher, and P. Dadam. Adaptive process management with ADEPT2. In *ICDE '05*, pages 1113–1114. IEEE Computer Press, 2005.

[31] J. Vanhatalo, H. Völzer, and J. Koehler. The refined process structure tree. *Data & Knowledge Engineering*, 68(9):793–818, 2009.

[32] E.W. Dijkstra. Notes on structured programming. pages 1–82, 1972.

[33] P. Dadam and M. Reichert. The ADEPT project: A decade of research and development for robust and flexible process support - challenges and achievements. *Computer Science - R & D*, 23(2):81–97, 2009.

[34] M. Minor, A. Tartakovski, D. Schmalen, and R. Bergmann. Agile workflow technology and case-based change reuse for long-term processes. *International Journal of Intelligent Information Technologies*, 4(1):80–98, 2008.

[35] H.A. Reijers and J. Mendling. Modularity in process models: Review and effects. In *BPM'08*, pages 20–35. LNCS 5240, Springer, 2008.

[36] J. Mendling, H.A. Reijers, and W.M.P. van der Aalst. Seven process modeling guidelines (7pmg). *Information & Software Technology*, 52(2):127–136, 2010.

[37] C. Combi and M. Gambini. Flaws in the flow: The weakness of unstructured business process modeling languages dealing with data. In *OTM Conferences (1)*, pages 42–59. LNCS 5870, Springer, 2009.

[38] L. Thom, M. Reichert, and C. Iochpe. Activity patterns in process-aware information systems: Basic concepts and empirical evidence. *Int. J. of Business Process Int. and Mgmt*, 4(2):93–110, 2009.

[39] S. Rinderle. *Schema Evolution in Process Management Systems*. PhD thesis, Ulm University, Germany, 2004.

[40] W.M.P. van der Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theor. Comput. Sci.*, 270(1-2):125–203, 2002.

[41] S. Rinderle-Ma, M. Reichert, and B. Weber. On the formal semantics of change patterns in process-aware information systems. In *ER'08*, LNCS 5231, pages 279–293, 2008.

[42] A. Lanz, U. Kreher, M. Reichert, and P. Dadam. Enabling process support for advanced applications with the AristaFlow BPM Suite. In *BPM'10 Demonstration Track*. Vol. 615 CEUR-WS.org, 2010.

[43] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[44] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[45] C. Li, M. Reichert, and A. Wombacher. Representing block-structured process models as order matrices: Basic concepts, formal properties, algorithms. Technical Report TR-CTIT-09-47, University of Twente, NL, 2009.

[46] G. F. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Pearson, 2005.

[47] P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[48] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., USA, 1993.

[49] A.M. Law. *Simulation modeling and analysis*. McGraw-Hill Higher Education, 2006.

[50] C. Li, M. Reichert, and A. Wombacher. A heuristic approach for discovering reference models by mining process model variants. Technical Report TR-CTIT-09-08, University of Twente, The Netherlands, March 2009.

[51] D.J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. CRC Press, 2004.

[52] A. Wombacher, P. Fankhauser, and E. Neuhold. Transforming bpel into annotated deterministic finite state automata for service discovery. *Web Services, IEEE International Conference on*, 0:316, 2004.

[53] B.F. van Dongen, A.K.A. de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A new era in process mining tool support. In *ICATPN*, pages 444–454. LNCS 3536, 2005.

[54] J. Dehnert and R. Rittgen. Relaxed soundness of business processes. In *CAiSE '01*, pages 157–170. LNCS 2068, Springer, 2001.

[55] C. Li, M. Reichert, and A. Wombacher. The MinAdept clustering approach for discovering reference process models out of process variants. *Int'l J. of Cooperative Information Systems*, 19(3 & 4):159–203, 2010.

[56] F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and M. La Rosa. Configurable workflow models. *Int. J. Cooperative Inf. Syst.*, 17(2):177–221, 2008.

[57] R. Lu and S.W. Sadiq. Managing process variants as an information resource. In *BPM'06*, pages 426–431, 2006.

[58] R. Lu and S. W. Sadiq. On the discovery of preferred work practice through business process variants. In *ER'07*, pages 165–180. Springer, 2007.

[59] J. Bae, L. Liu, J. Caverlee, L.J. Zhang, and H. Bae. Development of distance measures for process mining, discovery and integration. *Int. J. Web Service Res.*, 4(4):1–17, 2007.

[60] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM'01*, pages 313–320. IEEE, 2001.

[61] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM'02*, pages 721–724. IEEE Computer Society, 2002.

[62] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys*, 38(1):2, 2006.

[63] M. la Rosa, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Questionnaire-based variability modeling for system configuration. *Software and System Modeling*, 8(2):251–274, 2009.

[64] F. Gottschalk, T. A. C. Wagemakers, M. H. Jansen-Vullers, W.M.P. van der Aalst, and M. La Rosa. Configurable process models: Experiences from a municipality case study. In *CAiSE'09*, pages 486–500, 2009.

[65] J.S. Ash, M. Berg, and E. Coiera. Some unintended consequences of information technology in health care: the nature of patient care information system-related errors. *J. Am. Med. Inf. Ass.*, 11(2):104–112, 2004.

[66] C. Li, M. Reichert, and A. Wombacher. What are the problem makers: Ranking activities according to their relevance for process changes. In *ICWS'09*, pages 51–58. IEEE, 2009.

Appendix A. Properties of Block-structured Process Model

Let $S = (A, E, AT, ET, l)$ be a block-structured process model (cf. Def. 1). Then: S has the following structural properties:

1. S has a *unique start node*; i.e.; $\exists! s \in A : \forall (a_1, a_2) \in E : a_2 \neq s$.
   s is the only node with $AT(s) = \texttt{StartFlow}$.
2. S has a *unique end node*; i.e.; $\exists! e \in A : \forall (a_1, a_2) \in E : a_1 \neq e$.
   e is the only node with $AT(e) = \texttt{EndFlow}$.
3. Let A^{types} be defined as $A^{types} := \{a \in A | AT(a) \in types\}$, Then:
   - Each *non-split node* (excl. the end node) has exactly one outgoing precedence edge:
     $\forall a \in A \setminus A^{AndSplit \bigcup XorSplit \bigcup EndFlow}$:
     $\exists! e = (a_1, a_2) \in E$ with $a_1 = a \wedge ET(e) = \texttt{Precedence}$.
   - Each *non-join node* (excl. the start node) has exactly one incoming precedence edge:
     $\forall a \in A \setminus A^{AndJoin \bigcup XorJoin \bigcup StartFlow}$:
     $\exists! e = (a_1, a_2) \in E$ with $a_2 = a \wedge ET(e) = \texttt{Precedence}$.
   - Any *loop edge* links a StartLoop node with an EndLoop node:
     $\forall e = (a_1, a_2) \in E$ with $ET(e) = \texttt{Loop}$,
     $\Rightarrow AT(a_1) = \texttt{StartLoop} \wedge AT(a_2) = \texttt{EndLoop}$.

4. S is block-structured –i.e., the following properties hold:
   - Let $Splits, Joins \subset A$ be defined as follows:
     $Splits := A^{AndSplit \bigcup XorSplit}$, $Joins := A^{AndJoin \bigcup XorJoin}$.
     Then: There exists a mapping $join : Splits \to Joins$ with:
     - $s \in Splits, \Rightarrow s \prec join(s)$.
     - $join$ is a bijective mapping, i.e., $join(s_1) = join(s_2)$ for $s_1, s_2 \in Splits, \Rightarrow$ $s_1 = s_2 \wedge \forall j \in Joins : \exists s \in Splits : join(s) = j$.
     - Let $s \in Splits$:
       The subgraph induced by $\{s, join(s)\} \bigcup \{a \in A | s \prec a \wedge a \prec join(s)\}$ is a SESE, i.e., a subgraph with single entry and single exit node.
     - $s \in Splits \wedge AT(s) =$ `AndSplit(XorSplit)`,
       $\Rightarrow AT(join(s)) =$ `AndJoin(XorJoin)`.
   - There exists a bijective mapping $loop : A^{StartLoop} \to A^{EndLoop}$ with:
     - $ls \in A^{StartLoop}, \Rightarrow loop(ls) \prec ls$
     - $ls \in A^{StartLoop}, \Rightarrow$ The subgraph induced by
       $\{ls, loop(ls)\} \bigcup \{a \in A | loop(ls) \prec a \wedge a \prec ls\}$ is a SESE.
   - Blocks must not overlap, i.e., their nesting must be regular. Formally:
     $B_{starts} \equiv Splits \bigcup A^{EndLoop}$; $B_{ends} \equiv Joins \bigcup A^{StartLoop}$.
     Further, Let $block$ be a mapping,$block : B_{starts} \to B_{ends}$ with $block(s) = join(s)$ if $s \in Splits$ and $block(s) = loop^{-1}(s)$ if $s \in A^{EndLoop}$. Then: $s_1, s_2 \in B_{starts}$ with $s_1 \prec s_2 \prec block(s_1), \Rightarrow block(s_2) \prec block(s_1)$.

Appendix B. Proof of Theorem 1

Theorem 1 (cf. Section 3.1.3) states that we obtain a unique order matrix A for a process structure tree $T = (N, C, CT, E, l)$, i.e., for two nodes $a_i, a_j \in N$, $NCA(a_i, a_j)$ exists and is unique. The proof consists of 3 steps:

1. Based on the properties of process structure tree (cf. Theorem 2), we first prove that the indegree of any element in a process structure tree is less or equal to 1 (cf. Theorem 3).
2. We show that for any two connected nodes in a process structure tree, there exists exactly one path linking them (cf. Lemma 1).
3. Finally, we prove that for any two different nodes in a process structure tree, their nearest common ancestor exists and is unique (cf. Theorem 4).

We first discuss an important property of any process structure tree $T = (N, C, CT, E, l)$, namely that a subtree of T does not overlap with another different subtree of T. This property is described by Theorem 2. For a proof of Theorem 2, we refer to [31].

Theorem 2. *Let $T = (N, C, CT, E, l)$ be a process structure tree and let $T' = (N', C', CT', E', l')$ and $T'' = (N'', C'', CT', E'', l'')$ be two different subtrees of T. Then: T' does not overlap with T''; i.e., either T' is a subtree of T'', or T'' is a subtree of T', or the following property holds $((N' \bigcap N'' = \emptyset) \wedge (C' \bigcap C'' = \emptyset) \wedge (E' \bigcap E'' = \emptyset))$.*

Based on Theorem 2, we can obtain Theorem 3.

Theorem 3. *The indegree $in(e)$ of any element $e \in N \bigcup C$ in a process structure tree $T = (N, C, CT, E, l)$ is less or equal 1.*

Proof. Assume there exists an element $e \in N \bigcup C$ which has more than one predecessor; i.e., $\exists n_1, n_2, \ldots, n_i \in N \bigcup C : (n_1, e), (n_2, e), \ldots, (n_i, e) \in E$ and $n_x \neq n_y$ for $x, y \in \{1, \ldots, i\}$. Let $T(n_x)$ and $T(n_y)$ be two subtrees that result when using n_x and n_y as their root elements (with $x, y \in \{1, \ldots, i\}$, $x \neq y$) (cf. Section 3.1.1). Then, $T(n_x)$ and $T(n_y)$ contain element e since $(n_x, e) \in E$ and $(n_y, e) \in E$. However, since $n_x \neq n_y$ holds, $T(n_x)$ cannot be a subtree of $T(n_y)$, and vice versa. Therefore, $T(n_x)$ and $T(n_y)$ overlap, which contradicts to the property described in Theorem 2. Consequently, any element $e \in N \bigcup C$ maximally has one predecessor, i.e., $in(e) \leq 1$; $in(e) = 0$ holds if e is the root of T.

Based on Theorem 3, we can obtain Lemma 1.

Lemma 1. *For two connected nodes $a, b \in N \bigcup C$ in a process structure tree $T = (N, C, CT, E, l)$, there exists exactly one path connecting a with b.*

Proof. Let $a, b \in N \bigcup C$ be two elements of T. Assume that a and b are connected with $a \prec b$. Then there exists a sequence $< n_0, n_1, \ldots, n_i >$ with $n_0, \ldots, n_i \in N \bigcup C$, $n_0 = a$, $n_i = b$ and $(n_{k-1}, n_k) \in E$ for $k \in \{1, \ldots, i\}$. According to Theorem 3, $in(n_k) \leq 1$ holds $\Rightarrow$ The node which directly precedes n_k is unique and corresponds to n_{k-1}. Since this applies to all $k \in \{1, \ldots, i\}$, the path $< n_0, \ldots, n_i >$ is unique.

Finally, Theorem 4 describes the existence and uniqueness of the nearest common ancestor for any two different nodes in a process structure tree.

Theorem 4. *Taking two different nodes $a, b \in N$ in a process structure tree $T = (N, C, CT, E, l)$, their nearest common ancestor $NCA(a, b)$ exists and is unique.*

Proof. Let $a, b \in N$ be two different nodes and let further $c \in C$ be the nearest common ancestor of a and b. Since a and b are two different nodes, T contains at minimum two nodes and one connector. Consequently, there must be a root connector $r \in E$ with $r \prec a$ and $r \prec b$, since nodes constitute the leaves in the tree and cannot be a predecessor of any other tree element.

- Existence of c. Since each process structure tree has a unique root r, we obtain $r \prec a$ and $r \prec b$. According to Lemma 1, we can find two unique paths $< n_0, n_1, \ldots, n_i >$ with $n_0 = r$ and $n_i = a$, and $< n'_0, n'_1, \ldots, n'_j >$ with $n'_0 = r$ and $n'_j = b$ respectively. Let $\min(i, j)$ denote the minimum of i and j. Since $r = n_0 = n'_0$, there must be a k with $0 \leq k \leq \min(i, j)$ such that $n_0 = n'_0$, $n_1 = n'_1$, ..., $n_k = n'_k$. According to Def. 8 we obtain n_k as $NCA(a, b)$.
- Uniqueness of c. Assume there is another connector $c' \neq c$ with $c' \in C$, which is a nearest common ancestor of a and b. According to Def. 8 we obtain $c \not\prec c'$ and $c' \not\prec c$. Let r be the unique root of process structure tree T. Then we obtain $r \prec c$ and $r \prec c'$. Since c' is common ancestor of a and b, we obtain $c' \prec a$ and $c' \prec b$. Consequently, there are two different paths from r to a: $< r, \ldots, c, \ldots, a >$ and $< r, \ldots, c', \ldots, a >$. This contradicts to Lemma 1 $\Rightarrow$ c' cannot exist.

Appendix C. Heuristic Search Algorithm for Variant Mining

```
input  : A process model $S$; a collection of process variants $S_i, i = 1, \ldots, n$; allowed search distance $d$ ;
output : Resulting process model $S'$
1  Compute process structure tree $T_i = (N_i, C_i, CT_i, E_i, l_i)$ for each $S_i$, and let $AS = \bigcup_{i=1}^{n} N_i$   /* Define $AS$ as active activity set */;
2  $S_c = S$                                          /* Define initial candidate model */;
3  $t = 1$                                            /* Define initial search step */ ;
4  while $|AS| > 0$ and $t \leq d$ do                 /* Search condition */;
5      $S_{sib} = S_c$                                /* Set $S_c$ as initial $S_{sib}$ */ ;
6      Define $a_s$ as the selected activity ;
7      foreach $a_j \in AS$ do
8          $S_{kid}$ = FindBestKid($S_c$) ;
9          if Fitness($S_{kid}$) > Fitness($S_c$) then
10             if Fitness($S_{kid}$) > Fitness($S_{sib}$) then
11                 $S_{sib} = S_{kid}$ ;
12                 $a_s = a_j$ ;
13         else
14             $AS = AS \setminus \{a_j\}$            /* Best kid not better than its parent; */
15     if Fitness($S_{sib}$) > Fitness($S_c$) then
16         $S_c = S_{sib}$ ;                          /* Initiate next iteration */ ;
17         $AS = AS \setminus \{a_s\}$ ;
18     else
19         break ;
20     t = t+1 ;
```

Appendix D. Block Enumeration Algorithm

```
input  : A process model $S$, its process structure tree $T = (N, C, CT, E, l)$ and its order matrix $A_{n \times n}$
output : A set $BS$ with all possible blocks
1  Define $BS_x$ be a set of blocks containing blocks with $x$ activities. $x = (1, \ldots, n)$;
2  Define each activity $a_i$ as a block $B_i$, $i = (1, \ldots, n)$ ;
3  $BS_1 = \{B_1, \ldots, B_n\}$.                                  /* initial state */ ;
4  for $i = 2$ to $n$ do                                         /* Compute $BS_i$ */;
5      let j = 1; let k = i;
6      while $j \leq k$ do
7          k = i - j      /* A block containing $k$ activities can only be obtained by merging blocks containing $i$ and $j$ activities */;
8          foreach $(B_j, B_k) \in BS_j \times BS_k$ do      /* judge whether $B_j$ and $B_k$ can form a block */
9              merge = TRUE; if $B_j \bigcap B_k = \emptyset$ then        /* Disjoint? */
10                 foreach $(a_\alpha, a_\beta, a_\gamma) \in B_j \times B_k \times (N \setminus B_j \bigcup B_k)$ do
11                     if $A_{a_\alpha a_\gamma} \neq A_{a_\beta a_\gamma}$ then
12                         merge = FALSE    /* two blocks con merge only if they show same order relations to the activities out side the two blocks */;
13                         break ;
14             else
15                 merge = FALSE;
16             if merge = TRUE then
17                 $B_p = B_j \bigcup B_k$;
18                 $BS_i = BS_i \bigcup B_p$;
19         j = j + 1 ;
20 $BS = \bigcup_{x=1}^{n} BS_x$
```