On Measuring Process Model Similarity based on High-level Change Operations

Chen Li[1], Manfred Reichert[2], and Andreas Wombacher[3]

[1] Information System group, University of Twente, The Netherlands
lic@cs.utwente.nl
[2] Institute of Databases and Information System, Ulm University, Germany
manfred.reichert@uni-ulm.de
[3] Database group, University of Twente, The Netherlands
a.wombacher@utwente.nl

Abstract. For various applications there is the need to compare the similarity between two process models. For example, given the as-is and to-be models of a particular business process, we would like to know how much they differ from each other and how we can efficiently transform the as-is to the to-be model; or given a running process instance and its original process schema, we might be interested in the deviations between them (e.g. due to ad-hoc changes at instance level). Respective considerations can be useful, for example, to minimize the efforts for propagating the schema changes to other process instances as well. All these scenarios require a method to measure the similarity or distance between two process models based on the efforts for transforming the one into the other. In this paper, we provide an approach using digital logic to evaluate the distance and similarity between two process models based on high-level change operations (e.g. to add, delete or move activities). In this way, we can not only guarantee that model transformation results in a sound process model, but also ensure that related efforts are minimized.

1 Introduction

Business world is getting increasingly dynamic, requiring from companies to continuously adapt business processes as well as supporting *Process-Aware Information Systems* (PAISs) [3] in order to cope with the frequent and unprecedented changes in their business environment [19]. Organizations and enterprises need to continuously Re-engineer their Business Processes (BPR), i.e. they need to be able to flexibly upgrade and optimize their business processes in order to stay competitive in their market. Furthermore, PAISs should allow for process flexibility, i.e., it must be possible for users to deviate from the pre-defined process model at the instance level if required.

The pivotal research on process flexibility over the last years [1, 11] has provided the foundation for dynamic process change to reduce the cost of change

[1] Supported by the Netherlands Organization for Scientific Research (NWO) under contract number 612.066.512

in PAISs. Process flexibility denotes the capability to reflect externally triggered change by modifying only those aspects of a process that need to be changed, while keeping the other parts stable, i.e., the ability to change or evolve the process without completely replacing it [11]. To compare two process models is a fundamental task in this context. In particular, it becomes necessary to calculate the minimal difference between two process models based on high level changes. If we need to transform one model into another, for example, efforts can then be reduced and the transformation can go smoothly; i.e. we do not need to re-define the new process model from scratch, but only apply these high-level changes either at process type or process instance level. Several approaches like ADEPT [11], WASA [20] or TRAM [6], have emerged to enable process change support in PAIS (see [13] for an overview).

Based on the two assumptions that (1) process models are block-structured and (2) all activities in a process model have unique labels, this paper deals with the following fundamental research question:

Given two process models S and S', how much do they differ from each other in terms of high-level change operations? And what is the minimal effort, i.e. the minimal number of change operations needed to transform S into S'?

Clearly, our focus is on minimizing the number of high-level change operations needed to transform process model S into process model S'. Soundness of the resulting process model should be also not sacrificed. We apply the high-level change operations as described in [11, 19] in the given context. By considering high-level changes, we can distinguish our approach from traditional similarity measures like graph or sub-graph isomorphism [15]. Both only consider basic change primitives like insertion or deletion of single nodes and edges.

Answering the above research question will lead to better cost efficiency when performing BPR, since the efforts to implement the corresponding changes in the supporting PAIS are minimized. At process instance level, we can reduce the efforts to propagate process type changes to the running instances [13]. Finally the derived differences between original process model and its process instances can be used as a set of pure and concise logs for process mining [4].

In previous work, we have provided the technical foundation for users to flexibly change process models at both the process type and the process instance level. For example, users may dynamically *insert, delete* or *move* an activity at these two levels [11]. In addition, snapshot differential algorithms [7], known from database technology, can be used as a fast and secure method to detect the *change primitives* (e.g. to add or delete nodes and edges) needed to transform one process model into another.

Using this framework and snapshot differential algorithm, this paper applies Digital Logic in Boolean Algebra [14] to provide a new method to transform a process model into another one based on high-level change operations. This method does not only minimize the number of changes needed in this context, but also guarantees soundness of the changed process model, i.e. the process model remains correct when applying high-level change operations. We further provide two measures –*process distance* and *process similarity* –based on high-

level change operations, which indicate how costly it is to transform process model S into model S', and how different S and S' are.

The remainder of this paper is organized as follows: Sec. 2 introduces backgrounds needed for the understanding of this paper. In Sec. 3 we discuss reasons and difficulties for deriving high-level change operations. Sec. 4 describes an approach to detect the difference between two process models. Sec. 5 discuss related work. The paper concludes with a summary and outlook in Sec. 6.

2 Backgrounds

Let $\mathcal{P}$ denote the set of all correct process models. A particular *process model* $S = (N, E, \ldots) \in \mathcal{P}$ is defined as a well-structured Activity Net [11]. N constitutes a set of activities a_i and E is a set of precedence relations (i.e. control edges) between them. To limit the scope, we assume Activity Nets to be block structured. Examples are depicted in Fig 1.

We assume that a process change (i.e. Activity Net Change) is accomplished by applying a sequence of high-level change operations to a given process model S over time [11]. Such change operations modify the initial process model by altering the set of activities and/or their order relations. Thus, each application of a change operation results in a new process model. We define *process change* as follows:

Definition 1 (Process Change). *Let $\mathcal{P}$ denote the set of possible process models and $\mathcal{C}$ the set of possible process changes. Let $S, S' \in \mathcal{P}$ be two process models, let $\Delta \in \mathcal{C}$ be a process change, and let $\sigma = \langle \Delta_1, \Delta_2, \ldots \Delta_n \rangle \in \mathcal{C}^*$ 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 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 $S_i[\Delta\rangle S_{i+1}$ for $i \in \{1, \ldots n\}$.*

Examples of high-level change operations and their effects on a process model are depicted in Table 1. Issues concerning the correct use of these operations and related pre-/post- conditions are described in [11]. If some additional constraints are met, the high-level change operations depicted in Table 1 will be also applicable at process instance level. Although the depicted change operations are discussed in relation to our ADEPT framework [11], they are generic in the sense that they can be easily transferred to other process meta models as well. For example, the change operations in Table 1 can be also expressed by the life-cycle inheritance rule as used in the context of Petri Nets [16]. We are referring to ADEPT in this paper since it covers by far most high-level change operations and change patterns respectively when compared to other approaches [19]. It further has served as basis for representing our method.

A trace t on process model S denotes a valid execution sequence $t \equiv < a_1, a_2, \ldots, a_k >$ of activities $a_i \in N$ on S according to the control flow defined by S. All traces process model S can produce are summarized in trace

Table 1. *Examples of High-Level Change Operations*

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 $\mathcal{B}$. It is a conditional insert if [sc] is specified (i.e. [sc] = XOR)			
delete(S, X, [sc])	delete	X	S, [sc]
Effects on S: deletes activity X from S, i.e. X turns into a silent one. [sc] is specified ([sc] = XOR) when we block the branch with X, i.e. the branch which contains X will not be activated			
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 A and B. (it is a conditional insert if [sc] is specified)			
replace(S, X, Y)	replace	X	Y
Effects on S: replaces activity X by activity Y			

set $\mathcal{T}_S$. $t(a \prec b)$ is denoted as precedence relation between activities a and b in trace $t \equiv < a_1, a_2, \ldots, a_k >$ iff $\exists i < j : a_i = a \wedge a_j = b$. Here, we only consider traces composing 'real' activities, but no events related to silent activities i.e., activity nodes which contain no operation and exist only for control flow purpose, see Section 4.4. Finally, we will consider two process models as being the same if they are *trace equivalent*, i.e. $S \equiv S'$ iff $\mathcal{T}_S \equiv \mathcal{T}_{S'}$. The stronger notion of bi-similarity [5] is not required in our context.

3 High-level Change Operations

3.1 Complementary Nature of Change and Execution Logs

Most PAISs support ad-hoc deviations at instance level and record them in *change logs*. Thus, they provide additional information when compared to traditional PAISs which only record *execution logs* (which typically document the start and/or end time of activities). Change logs and execution logs document different run-time information on adaptive process instances and are not interchangeable. Even if the original process model is given, it will be not possible to convert the change log of a process instance to its execution log or vice-verse. As example, take the original and simplified patient treatment process as depicted in Fig. 1a: a patient is *admitted* to a hospital, where he first *registers*, then *receives treatment*, and finally *pays*. Assume that, due to an emergency situation, for one particular patient, we want to first start the treatment of this patient and allow him to register later during treatment. To represent this exceptional situation in the process model of the respective instance, the needed change would be to move activity *receive treatment* from its current position to a position parallel to activity *register*. This change leads to a new model S', i.e., $S[\sigma\rangle S'$ with $\sigma =< move(S,$

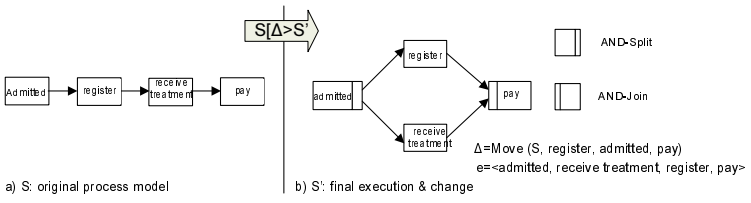

Fig. 1. Change Log and Execution Log are not Interchangeable

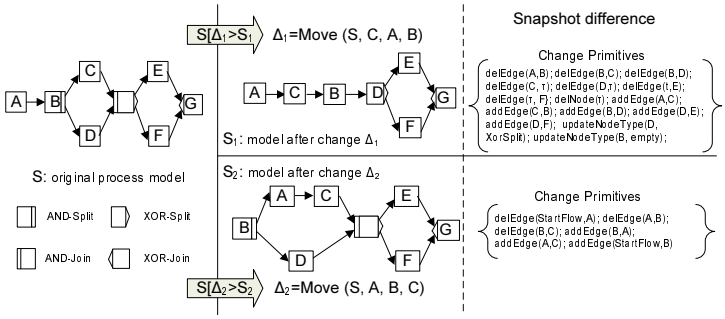

Fig. 2. High-level Change Operation and Corresponding change primitive

reveive treatment, *admitted*, *pay*) >. Meanwhile, the execution log e for this particular instance can be $e =<$ *admitted*, *receive treatment*, *register*, *pay* > (cf. Fig. 1b). If we only have process model S and its execution log, it will be not possible to determine this change because the process model which can produce such execution log is not unique. For example, a process model with the four activities contained in four parallel branches could produce this execution log as well. On the contrary, it is generally not possible to derive the execution log from a change log, because execution behavior of S' is also not unique. For example, a trace < *admitted*, *register*, *receive treatment*, *pay* > is also producible on S' as well. Consequently, change logs provide additional information when compared to pure execution logs.

3.2 Why Do We Need High-level Change Operations?

After showing the importance of change logs, we now discuss why we need high-level change operations rather than change primitives (i.e., low-level changes at edge and node level). Left side of Fig. 2 shows original process model S which consists of a parallel branching, a conditional branching, and a silent activity τ (depicted as empty node) connecting these two blocks. Assume that two different high-level change operations are applied to S resulting in models S_1 and S_2: Δ_1 moves activity `C` from its current location to the position between activities `A` and `B`, which leads to S_1 i.e., $S[\Delta_1\rangle S_1$ with $\Delta_1 = move(S, \texttt{C,A,B})$. Δ_2 moves `A` to the position between `B` and `C`, i.e. $S[\Delta_2\rangle S_2$ with $\Delta_2 = move(S, \texttt{A,B,C})$. Fig. 2 additionally depicts the change primitives representing snapshot differences between S and models S_1 and S_2, respectively. Using high-level change operations offers the following advantages:

1. High-level change operations guarantee soundness: i.e., application of a high-level change operation to a sound model S results in another sound model S' [11]. This also applies to our example from Fig. 2. By contrast, when applying one single change primitive (e.g., deleting an edge in S) soundness cannot be guaranteed anymore. Generally, if we delete any of the edges in S, the resulting process model will not be necessarily sound.
2. High-level change operations provide richer syntactical meanings than change primitives. Generally, a high-level change operation is built upon a set of change primitives which collectively represent a complex modification of a

process model. As example take Δ_1 from Fig. 2. This high-level change operation requires *15* change primitives for its realization (deleting edges, adding edges, deleting the silent activity, and updating the node types).

3. An important aspect, not discussed so far, concerns the number of change operations needed to transform model S into target model S'. For example, we need only *one* move operation to transform S to either S_1 or S_2. However, when using change primitives, migrating S to S_1 necessitates *15* change primitives, while the second change Δ_2 can be realized based on *6* change primitives. This example also shows that change primitives do not provide an adequate means to determine the difference between two process models. Thus the required number of change primitives cannot represent the efforts for process model transformations.

3.3 The Challenge to Derive High-level Change Operations

After sketching the benefits coming with high-level change operations, this section discusses challenges of deriving them. When comparing two process models, the change primitives needed for transforming one model into another can be easily determined by performing two snapshots and a delta analysis on them [7]. An algorithm to minimize the number of change primitives is given in [12]. However, when trying to derive the high-level change operations needed for model transformation, several challenges occur. As example consider Fig. 3:

1. When performing two delete operations on S (i.e., $\Delta_1 = \mathit{delete}(S, \mathtt{B})$ and $\Delta_2 = \mathit{delete}(S, \mathtt{C})$), we obtain a new model S'' (i.e., $S[\sigma\rangle S''$ with $\sigma =< \Delta_1, \Delta_2 >$), as well as an undetectable intermediate model S' with $S[\Delta_1\rangle S'$ and $S'[\Delta_2\rangle S''$. When examining the change primitives corresponding to each high-level change operation, we first need to add edge (`A,C`) after the first *delete* operation Δ_1, and remove this edge (`A,C`) when applying the second *delete* operation Δ_2. However, when performing a delta analysis for the original process model S and the resulting process model S'', the two change primitives (addEdge(`A,C`) introduced by the first *delete* operation and delEdge (`A,C`) introduced by the second one) jointly have no effect on the resulting process model S'', i.e., they cannot be detected by snapshot analysis. Consequently, deriving high-level change operations based on change primitives would be challenging because the change primitives required for every high-level change do not always appear in the snapshot differences between the original and resulting models. In Fig. 3, none of the two change primitive sets associated with Δ_1 or Δ_2 constitute a sub-set of the change primitive set associated with σ.
2. Even if there is just one high-level change operation, it will remain difficult to derive it with delta algorithm. For example, in Fig. 3 the delta algorithm shows that *15* change primitives are needed to transform S into S_1. However, the depicted changes can be also realized by just applying one high level move operation to S.

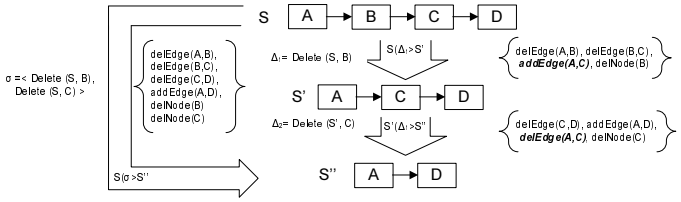

Fig. 3. Non-detectable Change Primitives

4 Detecting the Minimal Number of High-level Changes

In this section, we introduce our method to detect the minimal number of change operations needed to transform a given process model S into another model S'. As example, consider the process models S and S' in Fig. 4.

4.1 General Description of our Method

As mentioned in Section 1, the key issue of our work is to minimize the number of change operations needed to transform a process model $S = (N, E, \ldots) \in \mathcal{P}$ into another model $S' = (N', E', \ldots) \in \mathcal{P}$. Generally, three steps are needed (cf. Fig. 4) to realize this minimal transformation:

1. $\forall a_i \in N \setminus N'$: *delete* all activities being present in S, but not in S'. This first step transforms S to S_{same} (cf. Fig. 4b).
2. $\forall a_i \in N \bigcap N'$: *move* all activities being present in both models to the locations as reflected by S'. Regarding our example, this second step transforms S_{same} to S'_{same} (cf. Fig. 4c).
3. $\forall a_i \in N' \setminus N$: *insert* those activities being present in S', but not in S. As depicted in Fig. 4, the third step transforms S'_{same} to S' (cf. Fig. 4d).

Insertions and deletions deal with changes of the set of activities. Here, we can hardly do anything to reduce efforts (i.e., the number of required insert/delete operations): New activities ($a_i \in N' \setminus N$) must be added and obsolete activities ($a_j \in N \setminus N'$) must be deleted.

The focus of minimality can therefore be shifted to the use of the *move* operation, which changes the structure of a process model, but not its set of activities. Since a move operation logically corresponds to a delete followed by an insert operation, we can transform S_{same} to S'_{same} by maximally applying $n = |N \bigcap N'|$ move operations. Reason is that n move operations correspond to deleting all activities and then re-inserting them at their new positions. Correspondingly, n is the maximal number of change operations needed to transform one process model into another, both with same set of activities (S_{same} and S'_{same} in our example from Fig. 4). To measure the complete transformation from S to S', we formally define *process distance* and *process similarity* as follows:

Definition 2 (Process Distance and Process Similarity). *Let $S = (N, E, \ldots)$, $S' = (N', E', \ldots) \in \mathcal{P}$ be two process models. Let further $\sigma = \langle \Delta_1, \Delta_2, \ldots \Delta_n \rangle \in \mathcal{C}^*$ be a sequence of change operations transforming S into S' (i.e. $S[\sigma\rangle S'$). Then*

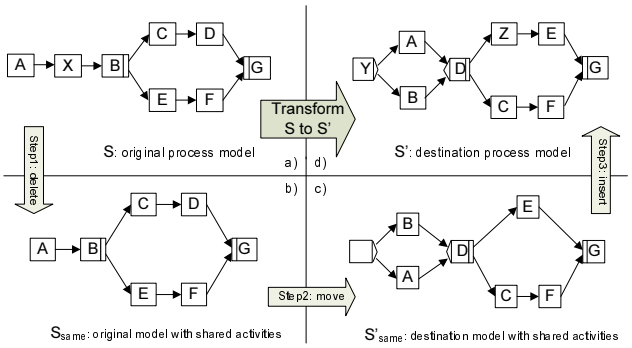

Fig. 4. Three Steps to Transform S into S'

the distance between S and S' is given by $d_{(S,S')} = min\{|\sigma| \mid \sigma \in \mathcal{C}^* \wedge S[\sigma\rangle S'\}$. Furthermore, process similarity between S and S' equals to $1 - \frac{d_{(S,S')}}{|N|+|N'|-|N \bigcap N'|}$, i.e., similarity equals to ((maximal number of changes - minimal number of changes) / maximal number of changes).

4.2 Determining Required Activity Deletions and Insertions

To accomplish Step 1 and Step 3 of our method, we have to deal with the change of the activity set when transforming S into S'. It can be easily detected by applying existing snapshot algorithms [7] to both S and S'. As described in Section 4.1, as first step we need to *delete* all activities $a_i \in N \setminus N'$ contained in S, but not in S'. Regarding our example from Fig. 4, we can derive as our first high-level change operation $\Delta_1 = delete(S, \mathtt{X})$. Similarly, activities contained in S', but not in S, are inserted in Step 3, after having moved the shared activities to their respective position in S' (S'_{same} respectively). The parameters of the insert operation, i.e. the predecessors and successors of the inserted activity, are just like how they appear in S'. In this way, we obtain the last two change operations for our example: $Insert(S, \mathtt{Y}, \mathtt{StartFlow}, \{\mathtt{A}, \mathtt{B}\})$ and $Insert(S, \mathtt{Z}, \mathtt{D}, \mathtt{E})$.

4.3 Determining Required Move Operations

We now focus on Step 2 of our method; i.e., to transform two process models with same activity set using move operations. Here, we can ignore the activities not contained in both S and S' (cf. 4.2). Instead, we consider the two process models S_{same} and S'_{same} respectively, as depicted in Fig. 4.

Determine the Order Matrix of a Process Model One key feature of our ADEPT change framework is to maintain the structure of the unchanged parts of a process model [11]. For example, if we delete an activity, this will neither influence the successors nor the predecessors of this activity, and also not their control relation. To incorporate this feature in our approach, rather than only looking at direct predecessor-successor relationships between two activities (i.e. control flow edges), we consider the transitive control dependencies between all pairs of activities; i.e., for every pair of activities $a_i, a_j \in N \bigcap N'$, $a_i \neq a_j$, their execution order compared to each other is examined. Logically, we check

execution orders by considering all traces a process model can produce (cf. Sec. 2). Results can be formally described in a matrix $A_{n\times n}$ with $n = |N \bigcap N'|$. Four types of control relations can be identified (cf. Def. 3):

Definition 3 (Order matrix). *Let $S = (N, E, \ldots) \in \mathcal{P}$ be a process model with $N = \{a_1, a_2, \ldots, a_n\}$. Let further $\mathcal{T}_S$ denote the set of all traces producible on S. Then: Matrix $A_{n\times n}$ is called* ***order matrix*** *of S with A_{ij} representing the relation between different activities a_i,$a_j \in N$ iff:*

- *A_{ij} = '1' iff ($\forall t \in \mathcal{T}_S$ with $a_i, a_j \in t \Rightarrow t(a_i \prec a_j)$)*
  If for all traces containing activities a_i and a_j, a_i always appears BEFORE *a_j, we denote A_{ij} as '**1**', i.e., a_i is predecessor of a_j in the flow of control.*
- *A_{ij} = '0' iff ($\forall t \in \mathcal{T}_S$ with $a_i, a_j \in t \Rightarrow t(a_j \prec a_i)$)*
  If for all traces containing activity a_i and a_j, a_i always appears AFTER *a_j, then we denote A_{ij} as a '**0**', i.e. a_i is successor of a_j in the flow of control.*
- *A_{ij} = '*' iff ($\exists t_1 \in \mathcal{T}_S$, with $a_i, a_j \in t_1 \wedge t_1(a_i \prec a_j)) \wedge (\exists t_2 \in \mathcal{T}_S$, with $a_i, a_j \in t_2 \wedge t_2(a_j \prec a_i)$)*
  *If there exists at least one trace in which a_i appears before a_j and at least one other trace in which a_i appears after a_j, we denote A_{ij} as '*****', i.e. a_i and a_j are contained in different parallel branches.*
- *A_{ij} = '-' iff ($\neg\exists t \in \mathcal{T}_S : a_i \in t \wedge a_j \in t$)*
  *If there is no trace containing both activity a_i and a_j, we denote A_{ij} as '**-**', i.e. a_i and a_j are contained in different branches of a conditional branching.*

We revisit our example from Fig. 4. The order matrices of S_{same} and S'_{same} are shown in Fig. 5. The main diagonal is empty since we do not compare an activity with itself. As one can see, elements A_{ij} and A_{ji} can be derived from each other. If activity a_i is a predecessor of activity a_j (i.e. $A_{ij} = 1$), we can always conclude that $A_{ji} = 0$ holds. Similarly, if $A_{ij} \in \{'*','-'\}$, we will obtain $A_{ji} = A_{ij}$. As a consequence, we can simplify our problem by only considering the upper triangular matrix $A = (A_{ij})_{j>i}$.

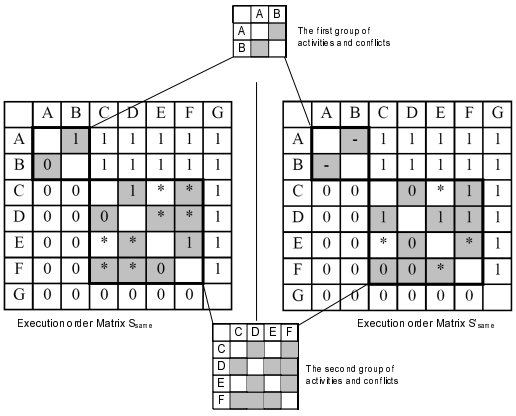

Fig. 5. Order Matrices of S_{same} and S'_{same} from Fig. 4

Under certain constraints, an order matrix A can uniquely represent the process model, based on which it was built on. This is stated by Theorem 1. Before giving this theorem, we need to define the notion of *substring of trace*:

Definition 4 (Substring of trace). *Let t and t' be two traces. We define t is a sub-string of t' iff $[\forall a_i,\ a_j \in t,\ t(a_i \prec a_j) \Rightarrow a_i,\ a_j \in t' \wedge t'(a_i \prec a_j)]$ and $[\exists a_k \in N\colon a_k \notin t \wedge a_k \in t']$.*

Theorem 1. *Let $S, S' \in \mathcal{P}$ be two process models, with same set of activities $N = \{a_1, a_2, \ldots, a_n\}$. Let further $\mathcal{T}_S$, $\mathcal{T}_{S'}$ be the related trace sets and $A_{n\times n}$, $A'_{n\times n}$ be the order matrices of S and S'. Then $S \neq S' \Leftrightarrow A \neq A'$, if $(\neg\exists t_1, t'_1 \in \mathcal{T}_S\colon t_1$ is a substring of $t'_1)$ and $(\neg\exists t_2, t'_2 \in \mathcal{T}_{S'}\colon t_2$ is a substring of $t'_2)$.*

According to Theorem 1, there will be a one-to-one mapping between a process model S and its order matrix A, if the substring constraint is met. A proof of Theorem 1 can be found in [8]. A detailed discussion of the sub-string restriction is given in Section 4.4. Thus, when comparing two process models, it is sufficient to compare their order matrices (cf. Def. 3), since a order matrix can uniquely represent the process model. This also means that the *differences of two process models* can be related to the *differences of their order matrices*. If two activities have different execution order in two process models, we will define the notion of *conflict* as follows:

Definition 5 (Conflict). *Let $S, S' \in \mathcal{P}$ be two process models with same set of activities N. Let further A and A' be the order matrices for S and S' respectively. Then: Activities a_i and a_j are* conflicting *iff $A_{ij} \neq A'_{ij}$. We formally denote this as $C_{(a_i,a_j)}$. $\mathcal{CF} := \{C_{(a_i,a_j)} \mid A_{ij} \neq A'_{ij}\}$ then corresponds to the set of all existing conflicts.*

Fig. 5 marks up differences between the two order matrices in grey. The set of conflicts is as follows: $\mathcal{CF} = \{C_{(\mathtt{A},\mathtt{B})}, C_{(\mathtt{C},\mathtt{D})}, C_{(\mathtt{C},\mathtt{F})}, C_{(\mathtt{D},\mathtt{E})}, C_{(\mathtt{D},\mathtt{F})}, C_{(\mathtt{E},\mathtt{F})}\}$.

Optimizing the Conflicts To come from S_{same} to S'_{same} (c.f. Fig. 4), we have to eliminate conflicts between these two models by applying *move* operations. Obviously, if there is no conflict for the two models, they will be identical. Every time we move an activity from its current position in S_{same} to the position it has in S'_{same}, we can eliminate the conflicts this activity has with other activities. For example, consider activity `A` in Fig. 4. If we move `A` from its position in S_{same} (preceding `B`) to its new position in S'_{same} (`A` and `B` are contained in two different branches of a conditional branching block), we can eliminate conflict $C_{(\mathtt{A},\mathtt{B})}$. As shown in the order matrices, moving `A` requires two steps. First, set the elements in the first row and first column of $A_{n\times n}$ (which corresponds to activity `A`) to empty, since `A` is moved away. Second, reset these elements according to the new order relation of `A`, when compared to the other activities from S'_{same}. So every time we move an activity, we are able to change the value of its corresponding row and column in the order matrices, i.e., we change these values corresponding to the original model to the values compliant with the target model. By doing

this iteratively, we can change all the values and eliminate all the conflicts so that we finally achieve the transformation from S_{same} to S'_{same}.

A non-optimal solution would be to move all the activities involved in the conflicts as set out by $\mathcal{CF}$, from their positions in S_{same} to the positions they have in S'_{same}. Regarding our example from Fig. 5, to apply this straightforward method, we would need to move activities A, B, C, D, E and F from their positions in S_{same} to the ones in S'_{same}. However, this naive method is not in line with our goal to minimize the number of applied change operations. For example, after moving activity A from its current position in S_{same} to the position it has in S'_{same}, we do not need to move activity B anymore, because after applying this change operation, there are no activities with which activity B still has conflicts.

Digital logic in Boolean algebra [14] helps to solve this minimization problem. Digital logic constitutes the basis for digital electronic circuit design and optimization. In this field, engineers face the challenge to optimize the internal circuit design given the required input and output signals. To apply such technique in our context, we consider each process activity as an independent input signal and we want to design a circuit which can cover all conflicts defined by $\mathcal{CF}$ (cf. Def 5). If activity a_i conflicts to activity a_j, we can either move one of them or both of them from the positions they have in S_{same} to the ones they have in S'_{same}. Doing so, the conflict will not exist any more. Reason is that every time we move an activity from the position it has in S_{same} to the position it has in S'_{same}, we reset the corresponding row and column of this activity in the order matrix. A conflict can be interpreted as a digital signal: When the two input signals a_i and a_j are both "true"(this means we do not move activity a_i and a_j), we cannot solve the conflict and the 'circuit' shall give an output signal of "false". If we apply this to all conflicts in $\mathcal{CF}$, we will obtain all "false" signals. Meanwhile, the "circuit" should be able to tell us what will result in a "true" output (i.e., the negative of all "false" signals). This "true" output represents which activities we need to move. Regarding our example from Fig. 5, given the set of conflicts $\mathcal{CF}$, our logic expression then is: $\overline{\mathtt{AB} + \mathtt{CD} + \mathtt{CF} + \mathtt{DE} + \mathtt{DF} + \mathtt{EF}}$.

The complexity for optimizing the logic expression is NP-Hard [14]. Therefore it is advantageous to reduce the size of the problem. Concerning our example, we can cut down the optimization problem into two groups: one with activities A and B, and conflict $C_{(\mathtt{A},\mathtt{B})}$; another one with activities C, D, E and F, and the following set of conflicts $\{C_{(\mathtt{C},\mathtt{D})}, C_{(\mathtt{C},\mathtt{F})}, C_{(\mathtt{D},\mathtt{E})}, C_{(\mathtt{D},\mathtt{F})}, C_{(\mathtt{E},\mathtt{F})}\}$. Such a division can be achieved in $O(n)$ time in the following three steps. Step 1: List all conflicting activities, and set every activity as a group. Step 2: If conflicting activities a_i and a_j (i.e., $C_{(a_i,a_j)}$) are contained in two different groups, merge the two groups. Step 3: Repeat Step 2 for all conflicts in $\mathcal{CF}$. After these three steps, we can divide the activities as well as the associated conflicts into several groups. Regarding our example, the optimization problem can be divided into two sub-optimization problems: $\overline{\mathtt{AB}}$ and $\overline{\mathtt{CD} + \mathtt{CF} + \mathtt{DE} + \mathtt{DF} + \mathtt{EF}}$. We depict this by the two small matrices in Fig. 5.

Optimizing logic expressions has been intensively discussed in Discrete Mathematics. Therefore we omit details here and refer to Karnaugh map [14] and

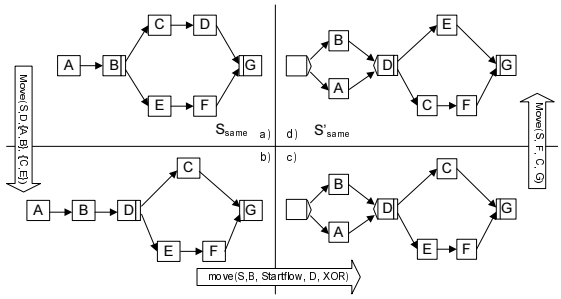

Fig. 6. Process Models After Every Move Operation

Quine-McCluskey algorithm [14]. We have implemented the latter in our proof-of-concept prototype. Regarding our example in Fig. 4, the two optimization results are $\overline{\texttt{AB}} = \bar{\texttt{A}} + \bar{\texttt{B}}$ for the first group and $\overline{\texttt{CD} + \texttt{CF} + \texttt{DE} + \texttt{DF} + \texttt{EF}} = \bar{\texttt{D}}\bar{\texttt{F}} + \bar{\texttt{C}}\bar{\texttt{E}}\bar{\texttt{F}} + \bar{\texttt{C}}\bar{\texttt{D}}\bar{\texttt{E}}$ for the second group. We can interpret this result as follows. For the second group, either we move activities `D` and `F`, or we move activities `C`, `E` and `F`, or we move activities `C`, `D` and `E` from their position in S_{same} to the positions they have in S'_{same}. Based on this we can transform S_{same} into S'_{same} since all conflicts are eliminated. As can be seen from the order matrices, if we change the value of the corresponding rows and columns of these activities in S_{same}, we can turn S_{same} into S'_{same}. Since we want to minimize the number of change operations, we can draw the conclusion that activities `D` and `F` must be moved. Same rule applies to the result of the first group. However, there is no difference whether to move either `A` or `B` since both operations count as one change operation. Here, we arbitrarily decide to move activity `B`.

So far we have determined the set of activities to be moved. The next step is to determine the positions where these activities need to be moved to. Operation $move(S, \texttt{X}, \mathcal{A}, \mathcal{B}, [sc])$ will be independent from other move operations (i.e., it does not matter in which order to move the respective activity) if its direct predecessors $\mathcal{A}$ and direct successors $\mathcal{B}$ do not belong to the set of activities to be moved. Regarding our example from Fig. 4, activity `F` satisfies this condition since its predecessor `C` and successor `G` are not moved. If this had not been the case, we would have to introduce silent activities to put the moved activity to its corresponding place in S'_{same}. For example, if we want to first move `B` to its position in S'_{same}, we will have to introduce a silent activity after `B` and before `C` and `E`. Only in this way, we can change the execution order of `B` to what it appears in S'_{same}. However, such silent activity will be not required if we first move activity `D` to the position it has in S'_{same}. A detailed discussion can be found in [11].

According to the position the moved activities have in S'_{same}, we can determine the parameters (i.e., the predecessors, successors and conditions) for every move operation. In S'_{same}, activity `D` has predecessors `A` and `B`, and successors `E` and `C`. So one move operations therefore is $move(S, \texttt{D}, \{\texttt{B},\texttt{A}\}, \{\texttt{C},\texttt{E}\})$. Similarly, we obtain the other two move operations: $move(S, \texttt{B}, \texttt{StartFlow}, \texttt{D}, XOR)$ and $move(S, \texttt{F}, \texttt{C}, \texttt{G})$. The intermediate process models resulting after every move operation are shown in Fig. 6. When comparing order matrices for each model in

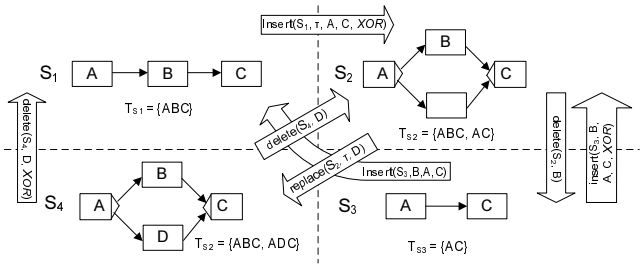

Fig. 7. The Influence of Silent Activity

Fig. 6, it becomes clear that every move operation changes the values of the row and the column corresponding to the moved activity.

4.4 Coping with Silent Activities

A silent activity is an activity which does not contain any operation or action, and which only exists for control flow purpose. There are two reasons why we do not consider silent activities in our similarity measure:

1. The appearance of a silent activity can be random. We can add or remove silent activities without changing the behavior of a process model, e.g., we can replace a control flow edge in a process model by one silent activity or even a block of silent activities without influencing process model behavior.
2. The existence of a silent activity also depends on other activities and is subject to change as other activities change. As example consider Fig. 2. When applying change Δ_1 to S, the silent activity τ is automatically removed after activity `C` is moved away.

There is one exception for which we need to consider silent activities. Consider the two process models S_1 and S_2 in Fig. 7. If we ignore the silent activity τ (depicted as an empty node) in S_2, and derive the order matrix of S_2, it will be the same as the one of S_1. Obviously, the two process models are not equivalent since the trace sets producible by them are not identical. More precisely, $\mathcal{T}_{S_2}$ contains one additional trace when compared to $\mathcal{T}_{S_1}$. In general, if one process model can produce additional traces, which are the sub-string of other traces (cf. Def.4), there must be some silent activities we cannot ignore. Or if the direct predecessor and direct successor of one silent activity constitute an XORsplit and XORjoin, we can also not ignore this silent activity (cf. S_2 in Fig. 7).

Fig. 7 shows several process model transformations based on high-level change operations. Here we can identify the difference between the two types of deletion: $delete(S_4, \mathtt{D})$ and $delete(S_4, \mathtt{D}, XOR)$ (cf. Fig. 7). The former one turns an activity into a silent one (transforming S_4 into S_2), while the latter one blocks the branch which contains activity `D` (transforming S_4 to S_1). When a branch is blocked, we do not allow the activities of the branch to become activated [16, 11]. Since process models S_1 and S_2 have same order matrix, purely comparing order matrices (cf. Sect. 4) would not be sufficient in the given situation. Reason is that here the order matrix does not uniquely represent the process model, since the sub-string constraint (cf. Def.4) of Theorem 1 is violated. To

		Figure 2			Figure 4			
		S	S_1	S_2	S	S_{same}	S'_{same}	S'
Figure 2	S	0 / 100%	1 / 86%	1 / 86%	4 / 50%	3 / 57%	3 / 57%	5 / 44%
	S_1		0 / 100%	2 / 71 %	4 / 50%	3 / 57%	3 / 57%	5 / 44%
	S_2			0 / 100%	5 / 38%	4 / 42%	3 / 57%	5 / 44%
Figure 4	S				0 / 100%	1 / 88%	4 / 50 %	6 / 40%
	S_{same}					0 / 100%	3 / 57%	5 / 44%
	S'_{same}						0 / 100%	2 / 78 %
	S'							0 / 100%

Fig. 8. Distances and Similarities of Different Process Models

extend our method such that it can uniquely represent a process model without the sub-string constraint, we must consider these special silent activities (i.e., a silent activity which is direct predecessor of an XORsplit and direct successor of an XORjoin) as well. They will appear in the order matrix and their execution orders compared with other activities will be documented.

However, the existence of a silent activity is still very much dependent on other activities, including the scenario described above. For example, if we delete `B` in S_2 as depicted in Fig. 7, we will transform S_2 into S_3, i.e., the silent activity will be simultaneously deleted when `B` is deleted. We can identify this situation by either examining the process model or the order matrix. In the process model, a silent activity τ can be automatically deleted if there is another silent activity τ' contained in the same block, but in another conditional branch (e.g., transforming S_2 to S_3). In the order matrix, we can automatically remove a silent activity τ if there is another silent activity τ' with same order relations to the rest of the activities as τ has.

In general, if a silent activity has an XORsplit as direct predecessor and an XORjoin as direct successor, we need to consider it when computing the order matrix of a process model. However, these silent activities can automatically be deleted when changing the process model. This requires us to perform additional checks on the process model or order matrix (as described above) after every change operation.

4.5 Summary

Taking our example from Fig. 4 (i.e., to transform S into S'), the following *six* change operations are required: $\sigma = \{delete(S, \texttt{X}), move(S, \texttt{F,C,G}), move(S, \texttt{D,\{A,B\},\{C;E\}}), move(S, \texttt{B,StartFlow,D}, XOR), insert(S, \texttt{Y,StartFlow,\{A,B\}}),$ and $insert(S, \texttt{Z,D,E})$ $\}$. Distance between the two models is *six* and similarity is *0.4* (cf. Def.2). To illustrate our method and these numbers in more detail, we compare the distances and similarities between the seven process models discussed so far: S, S_1 and S_2 from Fig. 2 and S, S_{same}, S'_{same} and S' from Fig. 4. Distance and similarity of two models are specified as *distance/similarity* in each corresponding cell in Fig. 8. As the transformation is commutable, we only fill in the upper triangle matrix. Taking Fig. 8, we can conclude:

1. Changing the activity set always leads to a modified distance. For example, $d_{(S_n, S'_{same})}$ always equals $d_{(S_n, S')} + 2$, where S_n stands for a process model other than S' or S'_{same} in Fig. 8. Reason is that S' contains two unique activities `Y` and `Z` when compared to S'_{same}, while the rest are identical.

2. If three process models S, S', and S'' have same activity sets, we will obtain $d_{(S,S'')} \leq d_{(S,S')} + d_{(S',S'')}$. It is easy to understand this because some activities could be moved twice when transforming S into S' and S' into S''.

5 Related Work

Various papers have studied the process similarity problem and provided useful results [17, 16, 21, 2]. In graph theory, graph isomorphism[15] and sub-graph isomorphism [15] are used to measure similarity between two graphs. Unfortunately, these measures usually only examine edges and nodes and cannot catch the syntactical issues of a PAIS (e.g., guarantee soundness of a process model, differentiate AND-Split and XOR-Split, and handle silent activities). Algorithms for measuring tree edit distances [2] shows similar disadvantages, i.e., syntactical issues of a PAIS are missing. In the database field, the delta-algorithm [7] is used to measure the difference between the two models. It extends the above mentioned approaches by assigning attributes to edges and nodes [12]. Still, it can only catch change primitives, and will further run into problems when considering high-level change operations. Regarding Petri-nets and state automata, similarity based on change is difficult to measure since these formalisms are not very tolerant for changes. Inheritance rules [16] are one of the very few techniques showing the transformation of a process model described as Petri-net. Trace equivalence is commonly used to compare whether two process models are similar or identical [5]. In addition, bisimulation [16, 18] extends trace equivalence by considering stronger notions. Also based on traces, [17] assign weights to each trace based on execution logs which reflect the importance of a certain trace. The edit distance [21] is also used to measure the difference between traces; the sum of them represents the differences of two models. Some similarity measures use two numbers (*precision* and *recall*) to evaluate the difference between process models S_1 and S_2 [17, 10]. None of these approaches measures similarity by a unique and commutative number, based on the effort for process transformation.

6 Summary and Outlook

We have provided a method to quantitatively measure the distance and similarity between two process models based on the efforts for model transformation. High-level change operations are used to evaluate the similarity since they guarantee soundness and also provide more meaningful results. We further applied digital logic in boolean algebra so that the number of change operations required to transform process model S into process model S' becomes minimal. Respective distance and similarity measures have already been applied in the filed of process mining [9].

Additional work is needed to enrich our knowledge on process similarity. As a first step, we will extend our method so that it is able to measure the similarity between process models with additional constructs (e.g., loopbacks [11]) and data flows. The next step will be to enrich the model with semantic relations between activities and to give weight for each change operation, so that the similarity measure can be further applied to practice.

References

1. P. Balabko, A. Wegmann, A. Ruppen, and N. Clément. Capturing design rationale with functional decomposition of roles in business processes modeling. *Software Process: Improvement and Practice*, 10(4):379–392, 2005.
2. P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.
3. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems.* Wiley & Sons, 2005.
4. C.W. Günther, S. Rinderle, M. Reichert, and W. M. P. van der Aalst. Change mining in adaptive process management systems. In *CoopIS'06, LNCS4275*, pages 309–326, 2006.
5. J. Hidders, M. Dumas, W.M.P. van der Aalst, A.H.M. ter Hofstede, and J. Verelst. When are two workflows the same? In *CATS '05:*, pages 3–11, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
6. M. Kradolfer and A. Geppert. Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *COOPIS '99*, page 104, Washington, DC, USA, 1999. IEEE Computer Society.
7. W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *VLDB '96*, pages 63–74, San Francisco, CA, USA, 1996.
8. C. Li, M. Reichert, and A. Wombacher. On measuring process model similarity based on high-level change operations. Technical Report TR-CTIT-07-89, University of Twente, 2007.
9. C. Li, M. Reichert, and A. Wombacher. Discovering reference process models by mining process variants. In *ICWS'08, to appear*, 2008.
10. S.S. Pinter and M. Golani. Discovering workflow models from activities' lifespans. *Comput. Ind.*, 53(3):283–296, 2004.
11. M. Reichert and P. Dadam. ADEPTflex -supporting dynamic changes of workflows without losing control. *Journal of Intelligent Info. Sys.*, 10(2):93–129, 1998.
12. S. Rinderle, M. Jurisch, and M. Reichert. On deriving net change information from change logs - the deltalayer-algorithm. In *BTW*, pages 364–381, 2007.
13. S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems: a survey. *Data Knowl. Eng.*, 50(1):9–34, 2004.
14. S.Brown and Z.Vranesic. *Fundamentals of Digital Logic with Verilog Design.* McGraw-Hill, 2003.
15. P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Addison-Wesley, 2005.
16. 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, uary.
17. W.M.P. van der Aalst, A.K.A. de Medeiros, and A.J.M.M. Weijters. Process equivalence: Comparing two process models based on observed behavior. In *Business Process Management (BPM'06)*, pages 129–144, 2006.
18. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.
19. B. Weber, S. Rinderle, and M. Reichert. Change patterns and change support features in process-aware information systems. In *CAiSE*, pages 574–588, 2007.
20. M. Weske. Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In *HICSS '01*, page 7051, Washington, DC, 2001.
21. A. Wombacher and M. Rozie. Evaluation of workflow similarity measures in service discovery. In *Service Oriented Electronic Commerce*, pages 51–71, 2006.