AUTOMATED PLANNING OF PROCESS MODELS: THE CONSTRUCTION OF SIMPLE MERGES

Research

Heinrich, Bernd, University of Regensburg, Regensburg, Germany,
bernd.heinrich@wiwi.uni-regensburg.de

Schön, Dominik, University of Regensburg, Regensburg, Germany,
dominik.schoen@wiwi.uni-regensburg.de

Abstract

Business processes evolve dynamically with changing business demands. Because of these fast changes, traditional process improvement techniques have to be adapted and extended since they often require a high degree of manual work. To reduce this degree of manual work, the automated planning of process models is proposed. In this context, we present a novel approach for an automated construction of the control flow structure simple merge (XOR join). This accounts for a necessary step towards an automated planning of entire process models. Here we build upon a planning domain, which gives us a general and formal basis to apply our approach independently from a specific process modeling language. To analyze the feasibility of our method, we mathematically evaluate the approach in terms of key properties like termination and completeness. Moreover, we implement the approach in a process planning software and apply it to several real-world processes.

Keywords: Business Process Management, Process planning, Automated planning, Control Flow Structures.

1 Introduction

Nowadays, as markets change, customer needs shift and new competitors evolve dynamically, companies must frequently (re)design their business processes to adapt them. At the same time, business processes span not only across departments of a single company but also across interorganizational collaborations of multiple companies, which makes process models even more complex. For instance, according to Heinrich *et al.* (2015), a European bank has modeled and (re)designed over 2,000 processes in different departments and areas in a project. These process models, which are composed of actions and corresponding control flow structures, have been modeled using the ARIS toolset and documented to support upcoming improvements and adaptations of processes. To keep the process models up-to-date, frequent (re)designs due to, for instance, the aforementioned challenges of today's business world have been necessary. Moreover, the authors state that employees of the bank as well as executives of other branches such as insurance and engineering highlighted the fact that process flexibility has become more and more important within the last decade. The reasons most frequently mentioned for this increased demand for flexibility are the growing frequency and complexity of such process (re)design projects, which involve a significant degree of manual work (cf. also Hornung *et al.*, 2007).

To ensure the required flexibility, several research fields in Business Process Management (BPM) striving to support modelers and business analysts via automatic techniques are of increasing importance. The research fields process mining as well as process model verification and validation assist the analyst in the process analysis phase (e.g., Wetzstein *et al.*, 2007). Automated (web) service composition can

be seen as part of the phases process implementation and process execution (Weber, 2007; Khan *et al.*, 2010). In the process modeling phase, which we will focus on in this paper, the goal of the research strand automated process planning is to enable the automated construction of process models using planning algorithms (Heinrich *et al.*, 2011; Heinrich *et al.*, 2009; Henneberger *et al.*, 2008; Hoffmann *et al.*, 2012; Lautenbacher *et al.*, 2009). Automated planning aims to increase the flexibility by definition (cf. van der Aalst, 2013) of the resulting process models and to (re)design process models - for processes that must be frequently (re)designed. The task of an automated construction of process models can be understood as a planning problem (Ghallab *et al.*, 2004) with the objective to arrange actions and control flow structures in an appropriate order based on both, an initial state as well as a non-empty set of goal states. Here, using a nondeterministic planning domain, allowing an abstract representation of process models, independent from a specific process representation language, enables a widespread use. A fundamental challenge for the automated planning of process models is to construct control flow structures which represent the control flow of a process (Russell *et al.*, 2006; van der Aalst *et al.*, 2003). More precisely, in order to plan more complex process models, not only a sequence of actions but also control flow structures like exclusive choice, parallel split or simple merge have to be constructed in an automated manner (cf., e.g., Heinrich *et al.*, 2015; Heinrich *et al.*, 2009; Hoffmann *et al.*, 2012).

The specific research goal of this paper is the automated construction of one of the most important control flow structures, namely simple merge. The simple merge serves as a join connector for two or more paths-segments (called branch) into one single subsequent branch (cf., e.g., Russell *et al.*, 2006; van der Aalst *et al.*, 2003) and thus reduces the size of process models. However, the construction of simple merges within an automated planning approach does not only focus on reducing the size of process models and thus – according to, for instance, Moreno-Montes de Oca *et al.* (2015), Mendling *et al.* (2010), Sánchez González *et al.* (2010), Cardoso (2007) – its complexity. More generally, we have to be able to construct minimal process models in our context by removing redundant and duplicate path-segments "as early as possible". Further, to increase the readability and understandability of process models especially for laymen, La Rosa *et al.* (2011) propose to use pattern-compounds (cf. also Gschwind *et al.*, 2008; Mendling *et al.*, 2010; Mendling *et al.*, 2007) as they represent well-formed and sound block-structured fragments of a process model. Simple merges as so called "join connectors" (Mendling *et al.*, 2010) therefore should only be constructed in accordance with the related "split connector" (i.e., the control flow structure exclusive choice). Following this, we aim to construct simple merges in accordance to existing exclusive choices.

The contributions of this paper are a formal definition of our planning domain and an algorithm for the automated construction of simple merges. In more detail:

❶ To follow the research field of automated process planning and thus to ensure a widespread use of our approach, we consider belief states (possibly infinite sets of world states that may exist before and after applying an action) as we address the planning of process models and thus abstract from individual process executions (cf. Ghallab *et al.*, 2004).

❷ When constructing simple merges in an automated manner, we have to construct *minimal* process models. Thus, we address nested simple merges in order to simplify process models by removing duplicate sequences of actions in several paths and construct simple merges "as early as possible".

❸ We further focus on constructing simple merges in *complete* in terms of merging all distinct paths of process models that can be merged.

❹ We have to consider block structures (cf. La Rosa *et al.*, 2011; Gschwind *et al.*, 2008; Mendling *et al.*, 2010; Mendling *et al.*, 2007) in order to increase the readability and understandability of process models by means of constructed simple merges.

In the following section, we discuss related work regarding our contributions ❶ to ❹. Thereafter we present the formal foundation for our approach in Section 3 and the running example, we will use to illustrate our approach, in Section 4. Sections 5 and 6 elaborate the major design decisions and the

proposed method to construct simple merges. In Section 7 we evaluate our approach before we conclude with a discussion, limitations of our work and an outlook to future research.

2 Related Work

Our work contributes to the research fields in BPM striving to support modelers and business analysts via automatic techniques. Especially, it focusses on the (1) automated planning of process models and is related to (2) process model complexity, (3) process modeling recommender systems and (4) process mining. Thus, we want to summarize and delimit existing research in these fields to our approach.

Ad (1): The research strand of automated planning of process models envisions the construction of process models by means of semantically annotated process elements and a semantic reasoning (Heinrich and Schön, 2015; Heinrich *et al.*, 2008; Heinrich *et al.*, 2011; Henneberger *et al.*, 2008; Hoffmann *et al.*, 2012; Lautenbacher *et al.*, 2009). Especially in Heinrich *et al.* (2008) and Henneberger *et al.* (2008) the challenges and the general planning approach is discussed. In this context, Heinrich *et al.* (2009) and Heinrich *et al.* (2015) propose an algorithm that copes with the construction of exclusive choices based on the determination of conditions. Their approach creates conditions that enable the construction of different outgoing branches of an exclusive choice, based on co-domain of belief state variables. However, they do not cope with simple merges. In contrast, Hoffmann *et al.* (2012) and Hoffmann *et al.* (2009) discuss the need of constructing simple merges (XOR joins) within their planning. However they do not provide any kind of algorithm or implementation for this problem (we ensured this by requesting an implementation from the authors). Summing up, an approach to construct simple merges in an automated manner is not presented so far (cf. contributions ❶ to ❹).

Ad (2): Following the idea of reducing the amount of manual work through automation, several works in the field of process model complexity address the appropriate construction of control flow structures as well. Process models need to be refactored based on rules regarding the envisioned structure of process models. Therefore, control flow structures need to be transformed and constructed respectively. Vanhatalo *et al.* (2008b), for example, present an approach for the automated completion of workflow graphs. Their method is based on a "well-behaved" graph with a single source (initial state) and a single sink (goal state). They aim at constructing simple merges only at the end of a process model (i.e., prior to the sink). Vanhatalo *et al.* (2009) and (2008a) introduce the concept of refined process structure trees and a method to represent workflow graphs in such a tree-based hierarchy of sub-workflows. They aim to identify these sub-workflows but not to consolidate equal sub-workflows (cf. ❷). Polyvyanyy *et al.* (2011) extend this approach and present an algorithm to transform a "multi-terminal graph" (MTG), which means, a graph that has at least one source and at least one sink, to a "two-terminal graph" (TTG), which means, a graph that has exactly one source and exactly one sink. They aim to connect the existing, multiple sinks of a MTG to one common single sink of the resulting TTG, but not at consolidating equal subtrees (i.e., the representation of equal sequences of actions in different paths; cf. contributions ❷ and ❹). Munoz-Gama *et al.* (2014) present an approach for the decomposition of process models. Their decomposition is based on so called "transition boundaries" or "place boundaries". That means that they identify subgraphs based on a single common action or belief state at the beginning of each subgraph. However, this is not a sufficient criterion for the construction of minimal process models and especially of nested simple merges (cf. ❷). Such nested simple merges do not necessarily require a single common *action* or *belief state* at the beginning of a subgraph. Further, none of the approaches in research field (2) copes with a nondeterministic planning domain and a state space with possibly infinite sets of world states, which is essential when addressing the automated planning of process models (cf. ❶).

Ad (3): The research strand of process modeling recommender systems focusses on issues like autocompletion of process models, finding (substructures of) process models in a repository suitable for a given problem definition or deciding where to start and stop modeling a process (cf., e.g., Fellmann *et al.*, 2015; Koschmider, 2007; Koschmider *et al.*, 2011) in order to reduce the manual modeling efforts. These works aim at suggestions on correct and fitting process fragments that can be used to complete

existing process models. In detail, during the construction of process models, recommender systems propose fitting process fragments (saved in a process model repository) based on (semantic) similarity measures of the fragments and the given problem definition, represented by, for instance, incomplete constructed process models at hand. This promising work, however, does not aim on constructing simple merges in an automated manner (cf. especially ❶ to ❸).

Ad (4): Besides these approaches, process mining aims at the partially automated reconstruction and redesign of process models based on event logs. Process mining allows discovering, checking and enhancing process models including workflow patterns (cf., e.g., Gaaloul *et al.*, 2005a) by means of event logs (cf., e.g., Accorsi *et al.*, 2012; van der Aalst *et al.*, 2012). As van der Aalst *et al.* (2012) explicate, process mining should support basic control flow structures. The authors stated that existing algorithms like the alpha algorithm (cf., e.g., van der Aalst *et al.*, 2004; Gaaloul *et al.*, 2005b; Gaaloul *et al.*, 2005a) are able to construct simple merges. However, these algorithms follow a local perspective by examining pairwise relations between two actions. Thus, they do not aim to construct simple merges *in complete* (cf. ❸) and may not provide minimal process models (cf. ❷). Further, to the best of our knowledge, we found no approach to construct simple merges in an automated manner that considers block structures (cf. ❹) for increasing the readability of the constructed process models. Moreover, current conversion algorithms (cf., e.g., Kalenkova *et al.*, 2015), that translate Petri Nets into Process Models (here: BPMN), do not deal with the completeness (cf. ❸) of the constructed simple merges. Moreover, as process mining aims to *re*construct *as-is* process models based on event logs, it does not cope with the ex-ante construction of *to-be* process models as it is addressed in this paper. Further, the field of process mining usually does not cope with a nondeterministic planning domain and a state space with possibly infinite sets of world states. So, the approaches for the construction of workflow patterns (as stated in e.g., Gaaloul *et al.*, 2005b; van der Aalst *et al.*, 2010), used in process mining, do not aim to address contribution ❶. To sum up, to the best of our knowledge, there exists no approach that addresses all contributions ❶ to ❹.

3 Fundamentals

As stated above, the construction of simple merges is a nondeterministic planning problem with belief states because we abstract from an individual process execution (Ghallab *et al.*, 2004). Using a nondeterministic planning domain which is independent from a particular process representation language enables a widespread use and guarantees compatibility with many existing approaches in the literature (e.g., Bertoli *et al.*, 2001; Bertoli *et al.*, 2006). A nondeterministic planning domain consists of a nondeterministic belief state-transition system which is defined in terms of its belief states, its actions, and of a transition function that describes how (the application of) an action leads from one belief state to possibly many belief states (acc. Bertoli *et al.*, 2006; Ghallab *et al.*, 2004). More formally, a belief state-transition system and (non-)determinism in state space are defined as follows:

Definition 1 (*Nondeterministic state-transition system*). A nondeterministic belief state-transition system is a tuple $\Sigma = (BS, A, R)$, where

- BS is a finite set of belief states. A belief state $bs \in BS$ contains a set BST of belief state tuples. A belief state tuple p is a tuple of a belief state variable $v(p)$ and a subset $r(p)$ of its predefined domain $dom(p)$, which we will write as $p:=(v(p),r(p))$.
- A is a finite set of actions. Each action $a \in A$ is a triple consisting of the action name and two sets, which we will write as $a:=(name(a), precond(a), effects(a))$. The set $precond(a) \subseteq BST$ are the preconditions of a and the set $effects(a) \subseteq BST$ are the effects of a.
- And $R: BS \times A \rightarrow 2^{BS}$ is the transition function. The transition function associates to each belief state $bs \in BS$ and to each action $a \in A$ the set $R(bs, a) \subseteq BS$ of next belief states.

According to this definition it is possible to represent possibly infinite sets of world states quite easily. Furthermore, it is a rather intuitive way – from a process modeling perspective – to represent certain preconditions and effects of actions.

Definition 2 ((*Non-)determinism in state space*). An action *a* is *applicable* in a belief state *bs* iff $|R(bs, a)| > 0$; it is *deterministic* (*nondeterministic*) in *bs* iff $|R(bs, a)| = 1$ ($|R(bs, a)| > 1$). If *a* is applicable in *bs*, then *R(bs, a)* is the set of belief states that can be reached from *bs* by performing *a*.

Based on both Definitions 1 and 2, a planning graph can be generated by means of different existing algorithms that progress from an initial belief state to goal belief states (see, e.g., Bertoli *et al.*, 2001; Bertoli *et al.*, 2006; Heinrich *et al.*, 2009; Heinrich *et al.*, 2011). In this paper, we primarily are extending these works by means of an approach to construct simple merges in an automated manner. With that said, we define our planning graph as follows:

Definition 3 (*planning graph*). A planning graph is an acyclic, bipartite, directed graph *G=(N, E),* with the set of nodes *N* and the set of edges *E*. Henceforth, the set of nodes *N* consists of two partitions: First, the set of flow nodes $Part_F$ (set *F* of flow nodes) which further contains two partitions, the set of action nodes $Part_A \subseteq Part_F$ (set *A* of actions) and the set of exclusive choice nodes $Part_{EC} \subseteq Part_F$ (set *EC* of exclusive choices), and second the set of belief state nodes $Part_{BS}$ (set *BS* of belief states). Each node $bs \in Part_{BS}$ is representing one distinct belief state in the planning graph. Each node $a \in Part_A$ is representing an action in the planning graph. Each node $ec \in Part_{EC}$ is representing an exclusive choice node in the planning graph. The planning graph starts with one initial belief state and ends with one to probably many goal belief states (with $Init \in BS$ and $Goal_j \in BS$).

When focusing on automated process planning, identical or very similar actions of a planning graph – as specified in Definition 3 – can be identified by using semantic concepts and automated reasoning (cf. e.g., Step 1, Heinrich *et al.*, 2015, p. 3). Such actions can then be identically (syntactically) labelled in the planning graph, which we will use in the following. However, our approach is not limited to the strand of automated process planning. In fact, for instance, within the research field of process mining, several works exist (see, e.g., Kindler *et al.*, 2006; van der Aalst *et al.*, 2010; Verbeek *et al.*, 2007) that use or construct graphs similar to planning graphs. Moreover, we envision to apply our approach to manually constructed process models by transferring them to the notions of a planning graph.

Given Definition 3, a planning graph consists of one to many paths. Here, a path is defined as follows:

Definition 4 (*path*). A path is sequence of nodes $n_p \in N$ starting with the initial belief state and ending in exactly one goal belief state.

Further, we define a branch as a subset of nodes regarding a particular path, starting with an *exclusive choice* node $ec \in Part_{EC}$:

Definition 5 (*branch*). A branch is a sequence of nodes $n_b \in N$ starting right after an exclusive choice node $ec \in Part_{EC}$ and ending in exactly one goal belief state. A branch therefore is a subset of nodes of a specific path.

4 Running example

We will use an excerpt of a real-world process taken from the order management of a European bank to illustrate our approach in the following. This excerpt of an order execution process model is represented by the planning graph (such a graph can be constructed with one of the approaches proposed by Bertoli *et al.*, 2006; Heinrich *et al.*, 2015; Heinrich *et al.*, 2009) shown in Figure 1. As illustrated, the order data must be entered and determined in a first step. After that, depending on the type of the security (conditions), a check must be stated and the order amount must be entered or calculated (in case of a stock order). Then, the plausibility of the order has to be proven and the order will get executed before the order is processed internally or externally and has to be assigned to a portfolio and documented. Finally, the order gets routed (note: the uppercase letters A-G are used to refer to the according XORs hereafter).

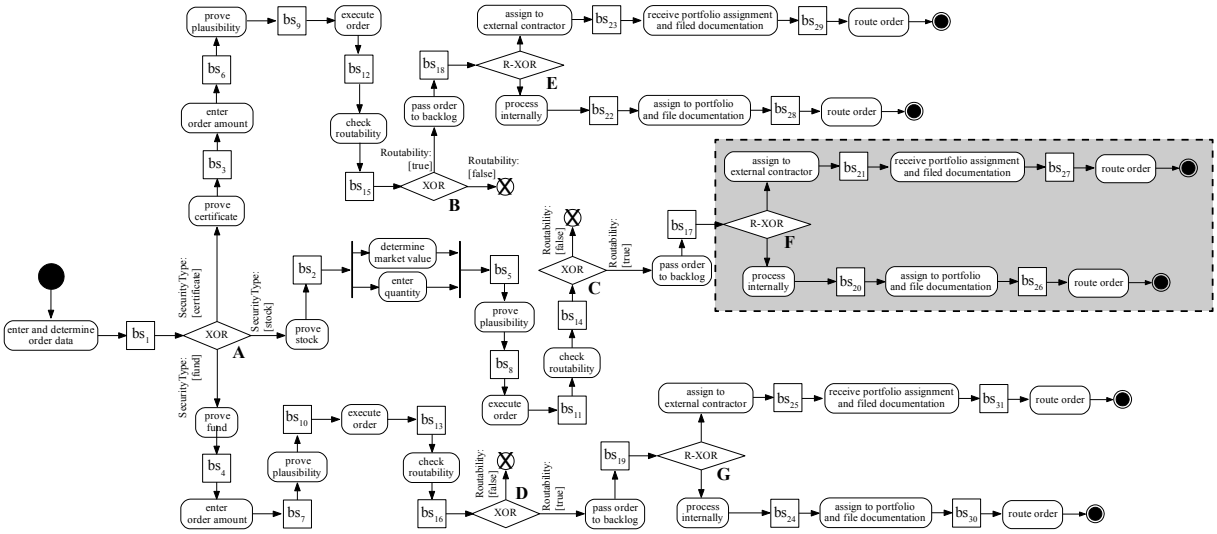

Figure 1. Planning graph of our running example

5 Design Process

In this section we will outline the major design decisions to address the contributions ❶-❹. These major design decisions are as follows:

1) *Traverse backwards:* To construct simple merges, we traverse the planning graph backwards, starting from the goal belief states. As the planning graph ends with one to many goal belief states, it is necessary to start the traversal with these goal belief states to identify all potentially mergeable paths and thus to cope with ❷ and ❸. The reason for traversing the planning graph backwards is that we are able to identify mergeable paths directly and do not need to traverse them completely from the initial state (like if we would use a forward traversal). Precisely, traversing backwards is effective because paths that can be combined by a simple merge need to *end* in both equal actions and equal control flow structures.
2) *Mark flow nodes with tokens:* To identify mergeable paths, we compare the flow nodes (partition $Part_F$) of the paths in a breadth-first manner, i.e., all flow nodes preceding the goal belief states are compared in the first iteration, all flow nodes preceding these in the second and so on. This approach, combined with 1), allows us to identify all mergeable paths directly (cf. ❷ and ❸), as they would be equal from a specific action in the path until the goal belief state. We use tokens to mark sets of equal flow nodes being part of different paths. For instance, given that the last flow node before the goal belief state is equal in three paths, we mark these flow nodes with the same token, otherwise with different ones. Continuing traversing backwards and annotating actions with tokens breadth-first allows us to recognize potentially mergeable paths even if they differ in subsequent iterations.
3) *Construct simple merges:* To assure that paths are merged only if they were split by an exclusive choice previously (cf. ❹), we construct simple merges when we reach an exclusive choice in the backward traversal. Here, the annotated tokens are used to identify mergeable paths.

Addressing design decision 2), we have to handle the need of identifying equal sequences of flow nodes within different paths. Thus, we define so-called *equality groups*:

Definition 6 *(equality group).* An *equality group* $eg_t = (n_1, \dots, n_m)$ is a tuple of flow nodes with the following properties:

- $n_i \in Part_F$ for all $i \in \{1, \dots, m\}$
- n_{i+1} succeeds n_i for all $i \in \{1, \dots, m-1\}$
- A token T_t denotes a flow node as member of the equality group eg_t

We denote the set of all equality groups by EG.

6 Method to Construct Simple Merges

In this section, we will elaborate and employ the above mentioned major design decisions to design a method that constructs simple merges and addresses the contributions ❶ to ❹.

6.1 Step 1: Merging one single exclusive choice

In the first step, we address to merge - as early as possible - two or more outgoing branches regarding a single exclusive choice. In this simple case, we conduct the following sub steps:

(1) Check all paths, starting from their corresponding goal belief states. Mark equal flow nodes with the same *equality token*, starting with the last flow node before each goal belief state.

(2) Continue traversing backwards, comparing the subsequent flow nodes of all paths. If the flow nodes of each considered equality group are still equal, simply add the same token to the flow nodes of these paths. If the flow nodes are different, add new tokens for each equality group of flow nodes. Consider that flow nodes could only be marked with the same additional token, if they are member of the same equality group so far, which means if their preceding (in the backward traversal) flow nodes are the same. We additionally add the tokens of previously marked flow nodes to each flow node for performance reasons with regard to later phases of our approach.

(3) If an exclusive choice is reached while traversing the branches backwards, we need to check the first flow node of all outgoing branches. Those branches that are marked with the same *equality token* are mergeable as they contain an equal sequence of flow nodes. Thus, they have to be merged with a *simple merge* just before the beginning of this equal sequence, which we will recognize as all contained actions are marked only with equal tokens. Further, the belief states of the merged branches have be joined (regarding our running example, the set union of *SecurityType: fund, SecurityType: certificate* and *SecurityType: stock*) to get the accurate belief state regarding the single subsequent path after the simple merge. If the branches are not marked with equal tokens they will be merged only before the goal belief state. The reached exclusive choice itself is marked with a new token and all common tokens of the merged branches, too.

   Note: If branches of an exclusive choice have different lengths, all branches, except the longest, stop the backward traversal at the exclusive choice. When the traversal of the longest branch reaches the exclusive choice, all outgoing branches are compared according to the upper description.

(4) Finally, we have to continue traversing backwards and marking all flow nodes with tokens. Further, the approach finally reaches the initial state and terminates.

To demonstrate this first step, we use the introduced order execution process model. For illustration purpose, we focus on the part of the process, which is framed in Figure 1. According to the example, both paths are marked with the token T_1, as both paths end with the equal action “route order” (sub step (1)). Further, the actions “assign to portfolio and file documentation” and “receive portfolio assignment and filed documentation” differ and hence are additionally marked with different tokens (T_2 resp. T_3) according to sub step (2). Then, reaching the exclusive choice, the outgoing branches are identified as mergeable, as both contain the token T_1. Thus, a simple merge is inserted before the action ”route order”, as this is the first flow node marked only with equal tokens in both branches (see rightmost graph in Figure 2). Further, all subsequent belief states (i.e., bs_3 and bs_4) have be joined.

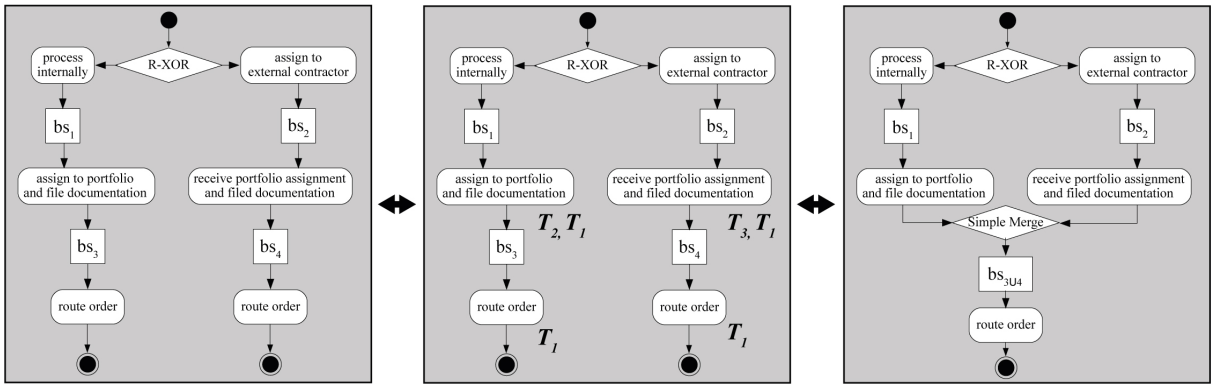

Figure 2. Constructing a simple merge regarding the running example

6.2 Step 2: Merging multiple nested exclusive choices

When process models become more complex, exclusive choices could occur in branches of other exclusive choices etc. Thus, our approach must be able to cope with *nested* exclusive choices (e.g. exclusive choices E, F and G are nested in exclusive choice A; cf. Figure 1). To address this challenging problem, we need to consider the main components of an exclusive choice, the outgoing branches and their conditions, and define a *choice construct* as follows:

Definition 7 *(choice construct).* The choice construct C of an exclusive choice is defined as a set of c ($c \in C$) where c is defined for each outgoing branch of this exclusive choice. Moreover, c is defined as the tuple *c:=(Condition, Tokens)* consisting of the condition of the corresponding branch (or null if the branch has no conditions) and the tokens of the first action of the branch succeeding the exclusive choice.

Based on this definition we are able to cope with *multiple nested* exclusive choices. To enable the merging of several paths containing exclusive choices with equal choice constructs – and therefore with equal outgoing branches and conditions – we need to *extend* the above sub step (3) as follows:

(3') As defined above in sub step (3), when reaching an exclusive choice, merge the outgoing branches. Additionally, mark the exclusive choice with its choice construct. Compare the exclusive choice with every other exclusive choice based on their choice constructs similar to comparing actions and mark them with the same token, when two or more choice constructs are equal.

The general specification in the first step above in combination with the straightforward extensions of Definition 7 and sub step (3') allows us to cope with the problem of merging multiple *nested* exclusive choices in outgoing branches now. For further comparison the tokens of the outgoing branches of a nested exclusive choice are not required anymore and thus, when marking subsequent flow nodes, only the token of the exclusive choice is needed.

To illustrate these extensions, we consider the whole planning graph of our running example as seen in Figure 1, containing multiple exclusive choices (denoted by capital letters). Initially, the procedure is equal to the excerpt in Figure 2 until the exclusive choices B, C and D are reached. When reaching each of the exclusive choices E, F and G, both outgoing branches are marked with the equality tokens T_1 and thus are mergeable before the action "route order". Further, exclusive choice F is marked with its choice construct, precisely *{(null, (T_2, T_1)), (null, (T_3, T_1))}* (cf. sub step (3')). As exclusive choices E and G contain the same outgoing branches as F, they are marked with the same choice construct. According to sub step (3'), all of those three exclusive choices get marked with T_4, as their choice constructs are equal.

When continuing the backward traversal, the exclusive choices B, C and D are reached and compared. They have to be marked with *{({(Routability, false)}, null), ({(Routability, true)}, (T_4, T_1))}* and their

outgoing branches can be merged. As there is only one outgoing branch that leads to the goal[1], no further merge is needed. Thereafter (cf. sub step (4)), the outgoing branches of exclusive choice A are checked. As they all contain the equal token T_5 (cf. the equal exclusive choices B, C and D) they will be merged before the action "prove plausibility" resulting in the graph shown in Figure 4.

6.3 Step 3: Merging exclusive choices within parallelization compounds

Regarding Mendling *et al.* (2010), process models should be "as structured as possible" (i.e., "every split connector matches a respective join connector of the same type"), which is related to contribution ❹. So far, the proposed approach is not able to provide this in case an exclusive choice is created within a parallelization compound (i.e., path segments surrounded by a parallel split node and a synchronization node). To solve this issue, the Definitions 3 to 5 must be extended to allow the representation of planning graphs containing parallelization compounds:

Definition 3' (*planning graph*). A planning graph is an acyclic, bipartite, directed graph $G=(N, E)$, with the set of nodes N and the set of edges E. Henceforth, the set of nodes N consists of two partitions: First, the set of flow nodes $Part_F$ (set F of flow nodes) which further contains four partitions, the set of action nodes $Part_A \subseteq Part_F$ (set A of actions), the set of exclusive choice nodes $Part_{EC} \subseteq Part_F$ (set EC of exclusive choices), the set of parallel split nodes $Part_{PS} \subseteq Part_F$ (set PS of parallel splits) and the set of synchronization nodes $Part_S \subseteq Part_F$ (set S of synchronizations), and second the set of belief state nodes $Part_{BS}$ (set BS of belief states). Each node $bs \in Part_{BS}$ is representing one distinct belief state in the planning graph. Each node $a \in Part_A$ is representing an action in the planning graph, each node $ec \in Part_{EC}$ is representing an exclusive choice node in the planning graph, each node $p \in Part_{PS}$ is representing a parallel split node in the planning graph and each node $s \in Part_S$ is representing a synchronization node in the planning graph. The planning graph starts with one initial belief state and ends with one to probably many goal belief states (with $Init \in BS$ and $Goal_j \in BS$).

Definition 4' (*path*). A path is sequence of nodes $n_p \in N$ either 1) starting with the initial belief state and ending in exactly one goal belief state or 2) starting with a parallel split node $p \in Part_{PS}$ and ending in the corresponding synchronization node $s \in Part_S$.

Definition 5' (*branch*). A branch is a sequence of nodes $n_b \in N$ starting right after an exclusive choice node $ec \in Part_{EC}$ and ending in exactly one goal belief state or a synchronization node $s \in Part_S$. A branch therefore is a subset of nodes of a specific path (see Definition 4').

Now, we are able to represent process models containing parallelization compounds by means of our planning domain. The upper part of Figure 3 shows an extension of our running example illustrating this. To enable the automated construction of simple merges within parallelization compounds we carefully extend the previously presented sub steps (1), (3) resp. (3') and (4) as follows:

(3'') When reaching a synchronization node, invoke the overall method (i.e., sub steps (1_{para}), (2), (3), (3'), (4_{para})) for all paths between the parallel split node and the synchronization node.

(1_{para}) Check all paths, starting from the final synchronization node. Mark equal flow nodes with the same *equality token*, starting with the last flow node before the synchronization node.

(4_{para}) Continue to traverse backwards and mark all flow nodes with tokens. Further, when finally reaching the initial parallel split node, return to the enclosing iteration.

We follow the above presented idea, used for merging nested exclusive choices, to allow merging branches that contain *nested* parallelization compounds and define a so called *parallel construct* based on the contained actions:

[1] The other outgoing branch leads to a so called flow final node, which terminates the process and is not needed to be merged.

Definition 8 *(parallel construct).* A parallel construct P of a parallel split is defined as a set of T_t $(Tt \in P)$ where T_t is denoted for each outgoing branch of this parallel split. Moreover, T_t is defined as the token of the first flow node of the branch succeeding the parallel split.

Further, when merging a nested parallelization compound, we compare it with every other parallelization compound in the currently analyzed equality group based on their parallel constructs (cf. sub step (2)) and mark them with the same token, when two or more parallel constructs are equal.

Regarding our running example (Figure 3), we traverse the parallelization compound backwards, starting with the synchronization node as stated in (1_{para}). Thereupon, "enter quantity" could be identified as equal in the two branches of the exclusive choice in sub step (2). When continuing the traversal and finally reaching the exclusive choice, these two branches will be merged as seen in the lower area of Figure 3 regarding (3). Regarding the previously defined sub steps (1) to (4), the parallel split will be marked with the parallel construct $\{T_2, T_4\}$ when finishing the traversal of the parallelization compound (cf. (4_{para})), as "determine real-time market value" gets marked with T_2 and "determine budget" gets marked with T_4. To sum up, we are now able to create minimal and block-structured (cf., e.g., La Rosa *et al.*, 2011; Mendling *et al.*, 2010) process models by constructing simple merges in complete.

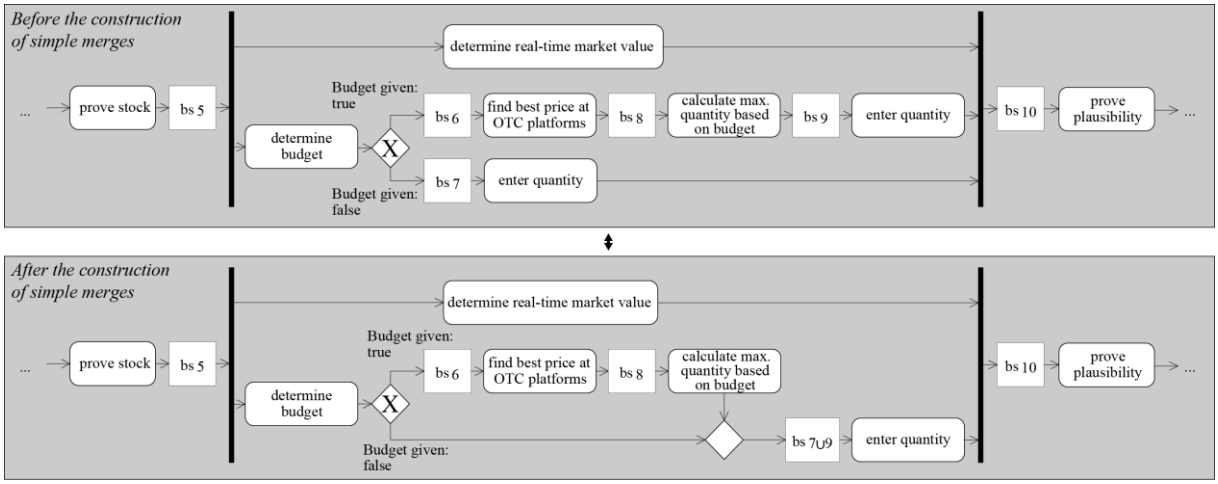

Figure 3. Unmerged and merged exclusive choice within a parallelization compound

7 Evaluation

In this section, we sketch the formal evaluation of our approach by mathematically proving that it provides minimal process models (i.e., simple merges are constructed "as early as possible"; cf. ❷), constructs simple merges if possible (i.e., completeness; cf. ❸) and terminates. We further briefly sketch its computational complexity and evaluate the results of our approach with respect to the construction of block structures (cf. ❹). Afterwards we evaluate the feasibility of the approach by means of a prototypical implementation and applying it to several real-world processes.

7.1 Formal evaluation of the approach

For the formal evaluation of our approach, we need to ensure that it terminates, identifies all mergeable paths (completeness), does not construct incorrect simple merges (correctness) and constructs simple merges as early as possible (i.e., is minimal). It is proven, that the approach meets all these criteria and its computational complexity is $O(n^5)$ in the number of goal belief states or sequential flow nodes of the longest path of the planning graph. This means, the algorithm is computationally efficient (cf., Arora and Barak, 2009; Cobham, 1965). It is further proven that for each exclusive choice, exactly one simple

merge is created (cf. ❹) so that the results of the approach are syntactically correct and fulfill the criteria of soundness and s-coverability. For the proof sketches see Appendices B and C[2].

7.2 Operational evaluation of the results

To assess the feasibility and applicability of our approach, the following questions are evaluated: (E1) Can our approach be instantiated in terms of a prototypical software implementation? (E2) Can it be applied in a practical setting and what is the output resulting from its application?

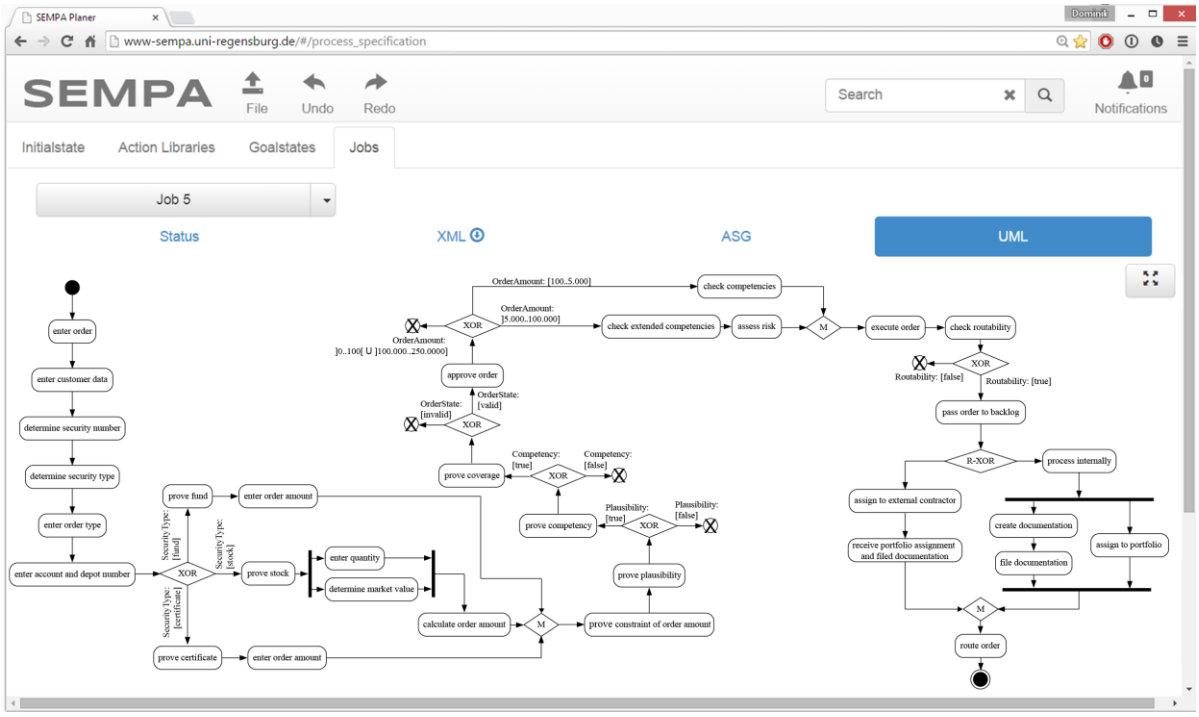

Figure 4. Screenshot of the constructed process model by means of our prototype

We integrated our approach in a web-based process planning tool (cf. E1; a demo version could be accessed at http://www-sempa.uni-regensburg.de/). The web application guides the user while populating the set of actions by providing descriptions of preconditions and effects and specifying the initial state and goal states. To test the implementation, persons other than the programmers analyzed the source code and several extreme value tests and unit tests have been performed. The implementation did not show any errors at the end of the test phase.

By means of the software implementation, we applied our approach to several real-world processes of a European bank, European insurance companies and an educational institution (cf. E2). Our running example is a part of one of these processes and addresses the order management of the bank (cf. Figures 1 and 5). To be able to apply our algorithm, the actions of former process models, used in the area of security order management had to be extracted and afterwards imported in our planning tool. About 200 actions including their preconditions and effects could be imported via an XML interface of ARIS, the bank's process modeling tool. We then reviewed them to validate, for instance, that their preconditions and effects were accurate noted. Finally, we specified the initial state and the intended goal states with the help of the employees of the bank's responsible department and planned the feasible process models using our planning tool. Hereby, two simple merges are constructed. The simple merge node that is firstly constructed merges the branches after "receive portfolio assignment and filed documentation"

[2] A version of this paper, containing all Appendices could be accessed via http://epub.uni-regensburg.de/33576/

and "assign to portfolio and file documentation". Its construction is straightforward as no nested exclusive choices need to be created. In the next step, the outgoing branches of the exclusive choices B, C and D are analyzed. As here only one branch leads to the goal and the other branch leads to a flow final node, no further simple merge needs to be created. As the previously merged exclusive choice occurs in this branch again, it is needed to consider the extensions for merging nested exclusive choices (cf. Section 5.2). In the last step, a simple merge node before "prove plausibility" merges the three different branches depending on the security type. Here, the extensions for nested exclusive choices (cf. Section 5.2) have to be considered again. In conclusion, only one action ("enter order amount") appears twice in the process model due to respecting pattern compounds (cf. contribution ❹).

Process number	Context	Num. of actions / states in the initial search graph	Num. of constructed simple merges	Num. of actions included by the simple merge	Num. of actions and control flow structures merged by the simple merge	Num. of merged paths	Run time in sec.
1	Project Mgmt.	17/15	1	4	1/0	3	<0.001
2	Project Mgmt.	25/18	1	1	7/2	2	<0.001
3	Project Mgmt.	26/22	3	8	8/1	4	0.002
4	Project Mgmt.	38/25	2	6	36/3	4	<0.001
5	Insurance Mgmt.	43/38	6	29	19/7	27	0.016
6	Insurance Mgmt.	54/44	8	33	18/8	3	0.006
7	Loan Mgmt.	40/31	3	25	12/3	3	0.003
8	Loan Mgmt.	57/43	4	14	11/4	20	0.013
9	Loan Mgmt.	122/69	3	54	4/0	0	0.024
10	Private Banking	278/189	25	82	69/16	6772	5.405
11	Human Res.	83/75	10	76	3/0	4	0.016

Table 1. Application of our approach in further real-use situations

As stated above, we applied the approach to further processes in different contexts of different firms. Table 1 shows the results of these applications (executed on an Intel Core i7-2600 3.40 GHz, Windows 7 64 Bit, Kernel Version 7601.22616, Java 8). The eleven analyzed processes of different application contexts include up to 278 actions and 189 states in the initial graph and are therefore of a small to a large size. The largest process model No. 10, for instance, contains actions conducted by several departments of the European bank and external service providers. Other processes are run by an insurance company, a mechanical engineering company (the context "Project management") and a university (the context "Human resources"). All process models include simple merges and for many process models nested control flow structures were created that merge a significant number of actions and states. This illustrates that nested simple merges are frequently used and relevant control flow structures. In all situations, our approach was fully applicable and generated correct and complete solutions. Thus the practical applicability of the approach is supported. The run time to construct simple merges varies from 0.001 up to 5.405 sec. and is thus very small.

Regarding economic aspects, Krause *et al.* (2013) present a quantitative evaluation, analyzing 18 process modeling projects from a financial service provider. Here, automated planning generates higher initial setup costs than manual process modeling, especially for analyzing and annotating actions. In contrast, the ongoing modeling costs are (as expected) much lower because of the support by the automated planning approach. Moreover, Krause *et al.* (2013) show that the use of automated planning of process models increases the contribution margin by about 20% which should cover its necessary higher initial investments. Automated planning should likely be even more valuable over a long term as Krause *et al.* (2013) considered only a short period of time. Alongside with this findings, applying the presented approach allows reducing the manual efforts when constructing process models and thus reduces the variable costs within the planning process. Hence it is supported that our approach is even more valuable for process models that need to be redesigned frequently as the initial costs can be amortized by savings

of parts of the variable costs that occur with each redesign (cf. also Heinrich *et al.*, 2015). Furthermore, the complexity of the process model could be decreased by reducing the amount of nodes within the process model, which then may lead to less errors during the execution of the process (cf. La Rosa *et al.*, 2011; Laue and Mendling, 2010; van der Aalst *et al.*, 2008).

8 Discussion and Conclusion

We propose a novel approach to construct the control flow structure simple merge in an automated manner and thus contribute to the research strand of automated process planning. Further, this work aligns to several research fields in BPM striving to support modelers and business analysts via automatic techniques. To abstract from individual process executions and to ensure a widespread use of our approach, we consider belief states (cf. ❶). We construct minimal (cf. ❷) and complete (cf. ❸) process models. Within this paper we additionally considered the construction of pattern compounds as proposed by e.g. La Rosa *et al.* (2011) in order to increase readability and understandability of the process models (cf. ❹). Our approach and the planning domain are formally noted and can therefore be well-defined and evaluated by means of mathematical proofs. This guarantees that key properties and envisioned contributions are met. Further, we discussed its applicability, feasibility and the results from the practical application by applying our approach (implemented in a process planning tool) to several real-world processes. In this context, we have tested that the construction of simple merges contributes to the automated planning of entire process models and thereby to reduce the amount of manual efforts within process (re-)design.

However, our research has some limitations that need to be addressed in the future. Constructing block structures (cf. ❹) may imply redundancies in the resulting process model that will not be merged in some cases (cf. action "enter order amount" in our running example). In some cases this might not be favorable, for instance, if the resulting process models are used only by domain experts in process modeling. In future research, this issue has to be addressed. To enable this, choice constructs of nested exclusive choices have to be compared in depth instead of only using their token to identify mergeability.

To complete the automated planning of entire process models, further (advanced) control flow structures should be constructed. Moreover, the work of Krause *et al.* (2013) should be followed-up to ensure a valuable usage of automated process planning in future real-world cases. For instance, it should be evaluated whether automated process planning makes it easier for laymen to construct correct and feasible process models. Further, it should be evaluated how the presented approach could be applied in other research strands such as process mining. As process mining approaches usually employ event logs to derive process descriptions (e.g., process graphs), we expect our approach to be beneficial especially regarding contributions ❷ and ❸ in this research strand, too. However, this idea needs to be evaluated in future research.

Further, it should be evaluated how process models modeled in a manual manner could be enriched to enable transferring them to planning graphs. Additionally, combined with assessing the semantic similarity of actions, our approach seems promising for supporting modelers as planning approaches (cf., e.g., Bertoli *et al.*, 2006; Heinrich *et al.*, 2008; Heinrich *et al.*, 2011) are already based on semantic annotations. For assessing the similarity of actions and subgraphs, works in the research fields of process management (cf., e.g., Ehrig *et al.*, 2007; Minor *et al.*, 2007; Montani *et al.*, 2015) and web service composition can be applied as especially the latter ones use input and output parameters as we do (cf., e.g., Dong *et al.*, 2004). Such works should be useable in the context of automated planning of process models as well. However, actions that are similar but not equal may not be merged in an automated manner but it might be possible to suggest modelers which actions may be merged after a modification. Such an extension of our approach provides a basis for promising advancements in the future.

Acknowledgements

The research was funded by the Austrian Science Fund (FWF): P 23546-G11.

References

Accorsi, R., Ullrich, M. and van der Aalst, W. (2012), "Aktuelles Schlagwort: Process Mining", in *Informatik Spektrum*, Vol. 35 No. 5, pp. 354–359.

Arora, S. and Barak, B. (2009), *Computational complexity: a modern approach*, Cambridge University Press.

Bertoli, P., Cimatti, A., Roveri, M. and Traverso, P. (2001), "Planning in nondeterministic domains under partial observability via symbolic model checking", in *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, Vol. 1, pp. 473–478.

Bertoli, P., Cimatti, A., Roveri, M. and Traverso, P. (2006), "Strong planning under partial observability", in *Artificial Intelligence*, Vol. 170 No. 4–5, pp. 337–384.

Cardoso, J. (2007), "Complexity analysis of BPEL Web processes", in *Software Process: Improvement and Practice*, Vol. 12 No. 1, pp. 35–49.

Cobham, A. (1965), "The intrinsic computational difficulty of functions", in *Logic, Methodology, and Philosophy of Science II*.

Dong, X., Halevy, A., Madhavan, J., Nemes, E. and Zhang, J. (2004), "Similarity search for web services", in *Proceedings of the Thirtieth International Conference on Very Large Data Bases: Toronto, Canada, Aug. 31-Sept. 3, 2004*, Vol. 30, Morgan Kaufmann, St. Louis, MO, pp. 372–383.

Ehrig, M., Koschmider, A. and Oberweis, A. (2007), "Measuring Similarity between Semantic Business Process Models", in Roddick, J.F. and Hinze, A. (Eds.), *Processdings of the Fourth Asia-Pacific Conference on Conceptual Modelling (APCCM 2007)*, Vol. 67, Ballarat, Victoria, Australia, pp. 71–80.

Fellmann, M., Zarvic, N., Metzger, D. and Koschmider, A. (2015), "Requirements Catalog for Business Process Modeling Recommender Systems", in Thomas, O. and Teuteberg, F. (Eds.), *Wirtschaftsinformatik Proceedings 2015, Osnabrück*, pp. 393–407.

Gaaloul, W., Alaoui, S., Baïna, K. and Godart, C. (2005a), "Mining Workflow Patterns through Event-data Analysis", in *Proceedings of the The 2005 Symposium on Applications and the Internet Workshops (SAINT-W'05)*, pp. 226–229.

Gaaloul, W., Baïna, K. and Godart, C. (2005b), "Towards Mining Structural Workflow Patterns", in Andersen, K., Debenham, J. and Wagner, R. (Eds.), *Database and Expert Systems Applications*, *Lecture Notes in Computer Science*, Vol. 3588, Springer Berlin Heidelberg, pp. 24–33.

Ghallab, M., Nau, D. and Traverso, P. (2004), *Automated Planning: Theory & Practice*, Morgan Kaufmann, San Francisco.

Gschwind, T., Koehler, J. and Wong, J. (2008), "Applying Patterns during Business Process Modeling. Business Process Management", in Dumas, M., Reichert, M. and Shan, M.-C. (Eds.), *Lecture Notes in Computer Science*, Vol. 5240, Springer Berlin / Heidelberg, pp. 4–19.

Heinrich, B., Bewernik, M.-A., Henneberger, M., Krammer, A. and Lautenbacher, F. (2008), "SEMPA - A Semantic Business Process Management Approach for the Planning of Process Models", in *Business & Information Systems Engineering (formerly Wirtschaftsinformatik)*, Vol. 50 No. 6, p. 445–460 (in German).

Heinrich, B., Bolsinger, M. and Bewernik, M. (2009), "Automated planning of process models: the construction of exclusive choices", in Chen, H. and Slaughter, S.A. (Eds.), *30th International Conference on Information Systems (ICIS)*, Springer, Phoenix, Arizona, USA, pp. 1–18.

Heinrich, B., Klier, M. and Zimmermann, S. (2011), "Automated Planning of Process Models –Towards a Semantic-based Approach", in *Smolnik, S; Teuteberg, F; Thomas, O. (Ed.): Semantic Technologies for Business and Information Systems Engineering: Concepts and Applications. Hershey: IGI Global*, pp. 169–194.

Heinrich, B., Klier, M. and Zimmermann, S. (2015), "Design of a Novel Approach to Construct Exclusive Choices", in *Decision Support Systems*, Vol. 78, pp. 1–14.

Heinrich, B. and Schön, D. (2015), "Automated Planning of context-aware Process Models", in Becker, J., vom Brocke, J. and Marco, M. de (Eds.), *Proceedings of the 23rd European Conference on Information Systems (ECIS), Münster, Germany, May 26-29*, Münster, Germany, p. Paper 75.

Henneberger, M., Heinrich, B., Lautenbacher, F. and Bauer, B. (2008), "Semantic-Based Planning of Process Models", in Bichler, M., Hess, T., Krcmar, H., Lechner, U., Matthes, F., Picot, A., Speitkamp, B. and Wolf, P. (Eds.), *Multikonferenz Wirtschaftsinformatik (MKWI)*, GITO-Verlag, München, pp. 1677–1689.

Hoffmann, J., Weber, I. and Kraft, F.M. (2009), "Planning@ sap: An application in business process management", in *Proceedings of the 2nd International Scheduling and Planning Applications woRKshop (SPARK'09)*.

Hoffmann, J., Weber, I. and Kraft, F.M. (2012), "SAP Speaks PDDL: Exploiting a Software-Engineering Model for Planning in Business Process Management", in *Journal of Artificial Intelligence Research*, Vol. 44 No. 1, pp. 587–632.

Hornung, T., Koschmider, A. and Oberweis, A. (2007), *A Rule-based Autocompletion Of Business Process Models*, available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.6389.

Kalenkova, A.A., van der Aalst, W.M.P., Lomazova, I.A. and Rubin, V.A. (2015), "Process Mining Using BPMN: Relating Event Logs and Process Models // Process mining using BPMN. Relating event logs and process models", in *Software and Systems Modeling*, pp. 1–30.

Khan, F.H., Bashir, S., Javed, M.Y., Khan, A. and Khiyal, Malik Sikandar Hayat (2010), "QoS Based Dynamic Web Services Composition & Execution", in *arXiv preprint arXiv:1003.1502*.

Kindler, E., Rubin, V. and Schäfer, W. (2006), "Process Mining and Petri Net Synthesis", in Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Eder, J. and Dustdar, S. (Eds.), *Business Process Management Workshops*, *Lecture Notes in Computer Science*, Vol. 4103, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 105–116.

Koschmider, A. (2007), *Ähnlichkeitsbasierte Modellierungsunterstützung für Geschäftsprozesse*, Univ.-Verl. Karlsruhe, Karlsruhe.

Koschmider, A., Hornung, T. and Oberweis, A. (2011), "Recommendation-based editor for business process modeling", in *Data & Knowledge Engineering*, Vol. 70 No. 6, pp. 483–503.

Krause, F., Bewernik, M.-A. and Fridgen, G. (2013), "Valuation of Manual and Automated Process Redesign from a Business Perspective", in *Business Process Management Journal*, Vol. 19 No. 1.

La Rosa, M., Wohed, P., Mendling, J., Ter Hofstede, A.H.M., Reijers, H.A. and van der Aalst, W.M.P. (2011), "Managing process model complexity via abstract syntax modifications", in *Industrial Informatics, IEEE Transactions on*, Vol. 7 No. 4, pp. 614–629.

Laue, R. and Mendling, J. (2010), "Structuredness and its significance for correctness of process models", in *Information Systems and e-Business Management*, Vol. 8 No. 3, pp. 287–307.

Lautenbacher, F., Eisenbarth, T. and Bauer, B. (2009), "Process model adaptation using semantic technologies", in *2009 13th Enterprise Distributed Object Computing Conference Workshops, EDOCW, Auckland, New Zealand*, pp. 301–309.

Mendling, J., Reijers, H.A. and Cardoso, J. (2007), "What Makes Process Models Understandable?", in Alonso, G., Dadam, P. and Rosemann, M. (Eds.), *Business Process Management*, *Lecture Notes in Computer Science*, Vol. 4714, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 48–63.

Mendling, J., Reijers, H.A. and van der Aalst, W.M.P. (2010), "Seven process modeling guidelines (7PMG)", in *Information and Software Technology*, Vol. 52 No. 2, pp. 127–136.

Minor, M., Tartakovski, A. and Bergmann, R. (2007), "Representation and Structure-Based Similarity Assessment for Agile Workflows", in Weber, R.O. and Richter, M.M. (Eds.), *Case-Based Reasoning Research and Development*, *Lecture Notes in Computer Science*, Vol. 4626, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 224–238.

Montani, S., Leonardi, G., Quaglini, S., Cavallini, A. and Micieli, G. (2015), "A knowledge-intensive approach to process similarity calculation", in *Expert Systems with Applications*, Vol. 42 No. 9, pp. 4207–4215.

Moreno-Montes de Oca, I., Snoeck, M., Reijers, H.A. and Rodríguez-Morffi, A. (2015), "A systematic literature review of studies on business process modeling quality", in *Information and Software Technology*, Vol. 58, pp. 187–205.

Munoz-Gama, J., Carmona, J. and van der Aalst, W.M. (2014), "Single-Entry Single-Exit decomposed conformance checking", in *Information Systems*, Vol. 46, pp. 102–122.

Polyvyanyy, A., Vanhatalo, J. and Völzer, H. (2011), "Simplified Computation and Generalization of the Refined Process Structure Tree", in *Proceedings of the 7th International Conference on Web Services and Formal Methods*, Springer-Verlag, Berlin, Heidelberg, pp. 25–41.

Russell, N., Ter Hofstede, A.H.M. and Mulyar, N. (2006), "Workflow ControlFlow Patterns: A Revised View", in *BPM Center Report BPM-06-22, http://bpmcenter.org/reports*.

Sánchez González, L., García Rubio, F., Ruiz González, F. and Piattini Velthuis, M. (2010), "Measurement in business processes. A systematic review", in *Business Process Management Journal*, Vol. 16 No. 1, pp. 114–134.

van der Aalst, W., Adriansyah, A., Medeiros, A. de, Arcieri, F., Baier, T., Blickle, T., Bose, J., van den Brand, P., Brandtjen, R. and Buijs, J. (2012), "Process Mining Manifesto", in *BPM 2011 Workshops, Part I, LNBIP 99*, pp. 169–194.

van der Aalst, W., Mylopoulos, J., Sadeh, N.M., Shaw, M.J., Szyperski, C. and Mendling, J. (2008), *Metrics for Process Models*, Vol. 6, Springer Berlin Heidelberg, Berlin, Heidelberg.

van der Aalst, W., Weijters, T. and Maruster, L. (2004), "Workflow mining: Discovering process models from event logs", in *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16 No. 9, pp. 1128–1142.

van der Aalst, W.M.P. (1998), "The application of Petri nets to workflow management", in *Journal of Circuits, Systems and Computers*, Vol. 8 No. 1, pp. 21–66.

van der Aalst, W.M.P. (2013), "Business process management: A comprehensive survey", in *ISRN Software Engineering*, Vol. 2013.

van der Aalst, W.M.P., Rubin, V., Verbeek, H.M., van Dongen, B.F., Kindler, E. and Günther, C.W. (2010), "Process mining: a two-step approach to balance between underfitting and overfitting", in *Software & Systems Modeling*, Vol. 9 No. 1, pp. 87–111.

van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B. and Barros, A.P. (2003), "Workflow Patterns", in *Distributed and Parallel Databases*, Vol. 14 No. 1, pp. 5–51.

Vanhatalo, J., Völzer, H. and Koehler, J. (2008a), "The Refined Process Structure Tree", in Dumas, M., Reichert, M. and Shan, M.-C. (Eds.), *Business Process Management*, *Lecture Notes in Computer Science*, Vol. 5240, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 100–115.

Vanhatalo, J., Völzer, H. and Koehler, J. (2009), "The refined process structure tree", in *Data & Knowledge Engineering*, Vol. 68 No. 9, pp. 793–818.

Vanhatalo, J., Völzer, H., Leymann, F. and Moser, S. (2008b), "Automatic Workflow Graph Refactoring and Completion. Service-Oriented Computing – ICSOC 2008", in Bouguettaya, A., Krueger, I. and Margaria, T. (Eds.), *Lecture Notes in Computer Science*, Vol. 5364, Springer Berlin / Heidelberg, pp. 100–115.

Verbeek, H.M.W., Basten, T. and van der Aalst, W.M.P. (2001), "Diagnosing Workflow Processes using Woflan", in *The Computer Journal*, Vol. 44 No. 4, pp. 246–279.

Verbeek, H.M.W., Pretorius, A.J., van der Aalst, W.M.P. and van Wijk, J.J. (2007), "Visualizing state spaces with Petri nets", in *Computer Science Report*, Vol. 7 No. 01.

Weber, I. (2007), "Requirements for implementing business process models through composition of semantic web services", in *Enterprise Interoperability II*, Springer, pp. 3–14.

Wetzstein, B., Ma, Z., Filipowska, A., Kaczmarek, M., Bhiri, S., Losada, S., Lopez-Cob, J.-M. and Cicurel, L. (2007), "Semantic Business Process Management: A Lifecycle Based Requirements Analysis", in *SBPM*.

Appendix A Pseudocode of the algorithm

```
1   procedure ALGORITHM(A)
2     AllChoiceNodes={}
3     AllParallelNodes={}
4     MARKACTIONS(A)
5   End
```

```
1   procedure MARKACTIONS(A)
2    ChoicePrecedings = {}
3    SubLength = A.length
4    while a := A.pop()
5     Precedings = {}
6     switch (a.type){
7       case JoinNode:
8         precedingTokens = PrecedingNode(a).tokens
9         a = MARKPARALLELS(a)
10        a.tokens.add(precedingTokens)
11        Precedings.add(PrecedingNode(a))
12        ADDUNIQUETOKEN(a)
13        AllParallelNodes.add(a)
14       break

15      case ChoiceNode:
16       if (PATHSNOTREADYYET())
17         break
18       endif
19       MARKCHOICE(a)
20       ADDUNIQUETOKEN(a)
21       forall b ∈ a.branches
22         a.tokens.add(b[0].tokens)
23       endfor
24       AllChoiceNodes.add(a)
25       MERGEPATHS(a)
26       ChoicePrecedings.add(PrecedingNode(a))
27      break

28      case Action:
29       Precedings.add(PrecedingNode(a))
30       EqualActions = {}
31       EqualActions.add(a)
32       forall b ∈ A
33         if (a == b && getUniqueToken(SucceedingNode(a)) ==
    getUniqueToken(SucceedingNode(b)))
34             Precedings.add(PrecedingNode(b))
35             EqualActions.add(b)
36             A.remove(b)
37         endif
38       endfor
39       uniqueToken=null
40       if (SubLength != EqualActions.length)
41             uID = GENERATEUNIQUEID()
42             uniqueToken = new Token(uID)
43       endif
44       forall b ∈ EqualActions
45         forall tok ∈ SUCCEEDINGNODE(b).tokens
46             b.tokens.add(tok)
47         endfor
48         b.tokens.add(uniqueToken)
49       endfor
50      break

51    if (Precedings.length > 0)
52      MARKACTIONS(Precedings)
53    endif
54  endwhile
```

```
55 if (ChoicePrecedings.length > 0)
56   MarkActions(ChoicePrecedings)
57 endif
58 end
```

```
1   procedure MarkChoice(a)
2     choiceConstruct = ChoiceConstruct.new
3     forall b ∈ a.branches
4       choiceConstruct.add({b.conditions, b.actions[0].tokens})
5     endfor
6     a.choiceConstruct = choiceConstruct
7     a.uniqueToken = null
8   End
```

```
1   procedure MarkParallels(node)
2     parallelConstruct = ParallelConstruct.New
3     forall a ∈ PrecedingNodes(node)
4         Branch = {}
5         while a != ParallelNode
6           if (a == JoinNode)
7             a = MarkParallels(a)
8           endif
9           Branch.add(a)
10          a = PrecedingNode(a)
11        endwhile
12        parallelConstruct.add(Branch)
13    endfor
14  a.parallelConstruct = parallelConstruct
15  a.uniqueToken = null
16  return a
17  End
```

```
1   procedure getUniqueToken(node)
2     switch node
3       case ChoiceNode
4         return node.uniqueToken
5       case ParallelNode
6         return node.uniqueToken
7       case Action
8         return node.tokens.last
9   end

10  procedure addUniqueToken(node)
11    boolean existsEqual = false
12    switch node
13      case ChoiceNode
14        forall choNode ∈ AllChoiceNodes
15          if (node.choiceConstruct == choNode.choiceConstruct)
16                  node.uniqueToken = getUniqueToken(choNode)
17                  existsEqual = true
18          endif
19        endfor
20      break

21      case ParallelNode
22        forall paraNode ∈ AllParallelNodes
23          if (node.parallelConstruct == paraNode.parallelConstruct)
24                  node.uniqueToken = getUniqueToken(ParaNode)
25                  existsEqual = true
26          endif
27        endfor

28    if !existsEqual
29      uID=generateUniqueID()
30      node.uniqueToken = token(uID)
```

```
31        node.tokens.add(token(uID))
32      endif
33    return
34    end
```

```
1   procedure MERGEPATHS (choiceNode)
2     NodeArray = {}
3     forall b ∈ choiceNode.branches
4       NodeArray.add(b.actions[0])
5     endfor
6     MERGEBRANCHES (NodeArray)
7   End
```

```
1   procedure MERGEBRANCHES (NodeArray)
2     TokenArray = new Array[NodeArray.length]
3     for (i=0; i < NodeArray.length; i++)
4       b = NodeArray[i]
5       tok = smallestToken(b) // at least T(0)
6       TokenArray[i] = tok
7       for (j=0; j < NodeArray.length; j++)
8         if (j != i)
9           c = NodeArray[j]
10          comp = max(b.SameTokens(c)) // biggest element in both b.Tokens and c.Tokens
11        if (tok < comp)
12          tok = comp
13        endif
14      endif
15    endfor
16    if (TokenArray[i] < tok)
17      TokenArray[i] = tok
18    endif
19  endfor
20  tok = max(TokenArray)
    // highest entry in Array.. will appear more than once
21  MergeArray = {}
22  IndexArray = {}
23  for (i = 0; i < TokenArray.length; i++)
24    if (TokenArray[i] == tok)
25      MergeArray.add(NodeArray[i])
26      IndexArray.add(i)
27    endif
28  endfor

29  NewMerge = CREATEMERGENODE(MergeArray, tok)
30 NodeArray.add(SucceedingNode(NewMerge))
31  NodeArray.deleteIndices(IndexArray)
32   if (NodeArray.length > 1)
33     MERGEBRANCHES (NodeArray)
34  endif
35 end
```

```
1   procedure CREATEMERGENODE(MergeArray,tok)
2     mergeNode = new MergeNode
3     forall node ∈ MergeArray
4       while (getUniqueToken(node)) != tok)
5         if !(SucceedingNode(node).isChoiceNode)
6           node = SucceedingNode(node) //If the algorithm reaches a ParallelNode, the
                                        SucceedingNode-function returns the succeeding Node of
                                        the JoinNode
7         if (getUniqueToken(node) == tok && node.isMergeNode)
8           mergeNode = node;
9           node = SucceedingNode(node)
10        endif
11      else
12        forall b ∈ node.branches
```

```
13          if (tok ∈ getAllTokens(b[0]))
14            node = b[0]
15          endif
16        endfor
17      endif
18    endwhile
19  endfor
20  forall b ∈ MergeArray
21    mergeNode.precedings.add(PrecedingNode(b)) //If preceding node of b is a mergeNode,
                                                  the PrecedingNode-function returns the
                                                  preceding nodes of the mergeNode
22  endfor
23  mergeNode.succeeding = MergeArray[0]
24  iteratingNode = MergeArray.pop()
25  while bs = getBelieveStateAfter(iteratingNode)
26    forall a ∈ MergeArray
27      bs = bs ∪ getBelieveStateAfter(a)
28    endfor
29    iteratingNode = SucceedingNode(iteratingNode)
30    for (i = 0; i < MergeArray.length; i++)
31      MergeArray[i] = SucceedingNode(MergeArray[i])
32    endfor
33  endwhile
34 return mergeNode
35 end
```

```
1   procedure PATHSNOTREADYYET(Node)
2      forall b ∈ Node.branches
3         if b[0].Tokens == {}
4            return true
5         endif
6      endfor
7   return false
```

Appendix B Mathematical evaluation of the algorithm

Appendix B.1 Termination

The ALGORITHM procedure terminates:

The lines 2 and 3 terminate obviously, so it suffices to show that the MARKACTIONS procedure starting in line 4 terminates.

The MARKACTIONS procedure terminates:

This is shown by proving that the number of iterations of each loop is finite, and that each statement of the algorithm (also the recursions in line 52 and line 56) terminates.

The while-loop starting in line 4 terminates:

- The statements in the while-loop are only executed for a finite number of elements, because *A* is finite.
- The statement in line 5 terminates obviously as it is a simple (set) operation.
- The switch-case statement starting in line 6 also terminates: The cases *JoinNode* and *ChoiceNode* terminate due to the fact that the procedures MARKPARALLELS, ADDUNIQUETOKEN, PATHSNOTREADYYET, MARKCHOICE, MERGEPATHS terminate (see Lemmata) and the rest of the statements are trivially terminating set operations. The case Action terminates as well: The statements in the lines 29-31 are again simple set operations. The for-loop starting in line 32 terminates because *A* is finite and the statements in lines 34-36 terminate trivially. The statements in the lines 39-42 terminate obviously. The for-loops starting in line 44 and 45 terminate since the sets *EqualActions* and *SucceedingNode().Tokens* are finite and the statements in line 46 and 48 are simple set operations.

The recursion in line 52 terminates because it is only invoked when the length of the set *Precedings* is non-zero. This happens only a finite number of times: The set *Precedings* starts empty (line 5). An element can only be added if there is a preceding node (cf. lines 11, 29, 34). This only occurs a finite number of times because every time the recursion is invoked, the algorithm goes one layer upward on the path. However, our paths are assumed to be finite.

Furthermore, the set *Precedings* is always finite because there are only a finite number of preceding nodes for every node and only the preceding nodes of a finite number of nodes are considered.

Finally, the recursion in line 56 terminates: It is only called upon when the length of the set *ChoicePrecedings* is non-zero. To prove that this occurs only a finite number of times, our argumentation is the following: The set *ChoicePrecedings* starts empty (line 2). An element can only be added if there is a preceding node (cf. line 26), and this only occurs a finite number of times because every time the recursion is invoked, the algorithm goes one layer upward on the path. However, our paths are assumed to be finite.

Additionally, the set *ChoicePrecedings* is always finite because there is only one preceding node for every node and only the preceding nodes of a single node are considered.

Lemma 1: MARKPARALLELS terminates:

To show that MARKPARALLELS terminates, it suffices to prove that the for-loop starting in line 3 terminates since the rest of the statements are obviously terminating. The for-loop is only called upon a finite number of times because there is only a finite number of preceding nodes for each node. Each iteration of the for-loop is finite, because the statements in the lines 4 and 12 are trivially terminating and the while-loop in line 6 gets called upon only a finite number of times. The reasoning for this is as follows: We may assume that *a* is not a *ParallelNode* (otherwise the while-loop doesn't get invoked at all). If *a*

is not a JoinNode, we move up one layer on our path because of line 10. This "moving-up" can only occur a finite number of times due to the assumption that our paths are finite. So we only have to consider the fact that *a* is a JoinNode. Then MARKPARALLELS gets invoked again, but with a node that is one layer upward compared to the initial invocation. Because our paths are assumed to be finite, also this moving upward must come to an end. Note that the lines 9 and 10 terminate obviously.

Lemma 2: ADDUNIQUETOKEN terminates:

Only the for-loops starting in the lines 5 and 13 need to be considered, the rest of the operations are simple set operations. These for-loops contain only simple set operations as well and are called upon only a finite number of times because the sets *AllChoiceNodes* and *AllParallelNodes* are finite: The sets start out empty (cf. lines 2 and 3 in ALGORITHM). An element can only be added to them via the MARKACTIONS primitive, in lines 24 resp. 13, and just one at most per iteration of MARKACTIONS. However, we have already seen that the MARKACTIONS procedure gets invoked only a finite number of times.

Lemma 2a: GETUNIQUETOKEN terminates:

Obvious.

Lemma 3: PATHSNOTREADYYET terminates:

This is clear since the for-loop in line 2 terminates, because there are only finitely many branches descending from each ChoiceNode.

Lemma 4: MARKCHOICE terminates:

To prove that MARKCHOICE terminates, it is sufficient to prove that the for-loop starting in line 3 terminates because the rest of the statements are trivially terminating. The for-loop is only invoked a finite number of times since there are only a finite number of branches after a choice node.

Lemma 5: MERGEPATHS terminates:

MERGEPATHS terminates, if MERGEBRANCHES in line 6 terminates, because the for-loop in line 3 is finite since every choice node only has a finite number of branches.

Lemma 6: MERGEBRANCHES terminates:

- The given *NodeArray* in line 1 is finite (as shown later). That causes the for-loops in lines 3 and 7 to be invoked only a finite number of times. All the other statements inside the for-loop starting in line 3 and also in the lines 20-22 are simple set operations. The next for-loop in line 23 terminates since *TokenArray* has the same length as *NodeArray* and the for-loop only contains simple set operations. Lines 29-31 terminate as CREATEMERGENODE terminates (see Lemma) and the other lines are simple set operations.
- The MERGEBRANCHES function terminates, if the recursive call in line 33 happens only a finite number of times, what will be shown in the following:
- The *NodeArray* is finite in line 33: It was finite in the initial call of the MERGEBRANCHES function (line 6 in MERGEPATHS), owing to the fact that a choice node only has a finite number of branches. It hasn't been changed before line 30. Furthermore, even though one node is being added to the *NodeArray* in line 30 and thus the length increases by one, in the next step in line 31 more than one node are being removed. This is caused by the fact that the length of *IndexArray* is greater than one. Hence, the length of *NodeArray* decreases in the lines 30 and 31 in total and therefore it decreases in every recursive call of the function, until its length is 1 and the recursion stops.

Lemma 7: CREATEMERGENODE terminates:

The for-loop starting in line 3 iterates only finitely many times, because *MergeArray* is finite as a subset of the finite set *NodeArray* (cf. lines 21, 25 in MERGEBRANCHES).

- The while-loop starting in line 4 iterates finitely many times as well: When the unique token of the current node is not *tok*, one of the following cases can occur:
  - The succeeding node is not a choice node: In this case the algorithm traverses further to the next succeeding node.
  - The succeeding node is a choice node: Then one of the first nodes of the branches of the choice node is marked with the token *tok*, and the algorithm also proceeds to this succeeding node that is marked with *tok* (see line 14). Because our paths are finite, this moving on can only occur a finite number of times, until the algorithm reaches a node whose highest token (i.e. uniqueToken) is *tok*.
- The operations within the while-loop starting in line 4 terminate, as they are simple set operations except for the for-loop starting in line 12, which terminates because a choice node only has finitely many branches.

These statements patched together allow us to conclude that the for-loop starting in line 3 terminates.

The for-loop starting in line 20 terminates, because *MergeArray* is finite, as we have already seen, and line 21 is a trivially terminating set operation.

The lines 23 and 24 terminate obviously.

- The for-loop starting in line 26 terminates, because line 27 is a simple set operation and *MergeArray* is finite (it was finite before and no element has been added).
- Line 29 is trivially terminating.
- The for-loop starting in line 30 terminates, because it only contains a simple set operation and we know *MergeArray* is finite.
- The while-loop in line 25 gets invoked only a finite number of times, because in each iteration one proceeds to a succeeding node in line 29 and our paths are assumed to be finite.

These statements allow us to conclude that the while-loop from line 25 terminates. This finishes the proof of the termination of CREATEMERGENODE.

Appendix B.2 Completeness and correctness (proof sketch)

The algorithm identifies all mergeable paths and does not create wrong simple merges:

We prove this theorem by proving the following two statements:

1) A node *a* can only get the same unique token as a node *b*, if node *a* is in the same equality groups as node *b* (short and in quantifiers: $(UniqueToken(a) = UniqueToken(b)) \Rightarrow (\forall\, eg \in EG: a \in eg \iff b \in eg)$).This leads to no incorrect merges being done.
2) If two nodes *a* and *b* are in the same equality groups, then *a* and *b* get the same token (short and in quantifiers: $(\forall\, eg \in EG: a \in eg \iff b \in eg) \Rightarrow (UniqueToken(a) = UniqueToken(b))$). This leads to all correct merges being done.

Concerning 1):

Let us assume there exist two nodes *a* and *b* which have the same unique token.

Case a): The nodes *a* and *b* are choice nodes (parallel nodes)

A choice node (parallel node) gets its token via ADDUNIQUETOKEN. As seen in lines 5-10 (lines 13-18) of ADDUNIQUETOKEN, such a node gets the same token as an existing node, if and only if they

have the same choice constructs (parallel constructs), otherwise they get a new unique token (line 19-23). Having the same choice constructs (parallel constructs) means that they have the same following branches with the same conditions, as can be seen in the MARKCHOICE (MARKPARALLELS) procedure. Thus *a* and *b* are in the same equality groups.

Case b): The nodes *a* and *b* are both actions

An action gets its unique token in line 48 in the MARKACTIONS primitive. Because of the structure of the algorithm, the nodes *a* and *b* having the same unique token means that either

b1) "*a* and *b* on the same layer"

Both *a* and *b* are in the same *EqualActions*-set. This means that the actions have the same effect and their succeeding nodes are equal. So they are in the same equality groups.

b2) "*a* and *b* not on the same layer"

The actions *a* and *b* are not in the same *EqualActions*-set, but they have the same unique token nevertheless. Without loss of generality we may assume that *a* got its tokens first (let us say, in iteration i_a) and the tokens of *b* were assigned later (in iteration i_b). Then in all the iterations between i_a and i_b, *SubLength* was equal to *EqualActions.length* (otherwise a new unique token would have been generated, cf. lines 40-43 in MARKACTIONS). This means that *a* and *b* are in the same equality groups.

Case c): The nodes *a* and *b* are two different kinds of nodes

If a node is a choice node (parallel node), it only gets the same unique token as existing choice nodes (parallel nodes), or it gets a new unique token, but never the same unique token as an already existing action node, or a parallel node (choice node) (see proof of case a) for details). On the other hand, later the same tokens that a choice node (parallel node) already possesses may be assigned to an action node, but in this case, the argumentation of case b2) shows that then the action node and the choice node (parallel node) are in the same equality groups.

Concerning 2):

We may assume that the nodes *a* and *b* are in the same equality groups.

Case a): The nodes *a* and *b* are choice nodes (parallel nodes)

When *a* and *b* are in the same equality groups, their following branches and those conditions are the same. This results in their choice constructs (parallel constructs), which are constructed in MARKCHOICE (MARKPARALLELS) being equal. Thus the lines 5-10 (lines 13-18) in ADDUNIQUETOKEN result in *a* and *b* getting the same token.

Case b): The nodes *a* and *b* are both actions

They are in the same equality groups, which can mean either

b1) "*a* and *b* on the same layer"

The actions *a* and *b* have the same effect and their succeeding nodes are equal. Then both, *a* and *b*, are added to the array *Precedings* in the same iteration (cf. lines 29, 34 in MARKACTIONS). So there is an invocation of MARKACTIONS(*A*) where *A* contains both *a* and *b*. When either *a* or *b* gets added to the array *EqualActions*, the other one gets as well (see lines 31-33, 35 in MARKACTIONS). Lines 44 and onward show that this results in marking *a* and *b* with the same token.

b2) "a and b not on the same layer"

Without loss of generality we may assume that a gets its tokens first (let us say, in iteration i_a) and the tokens of b are assigned later (in iteration i_b). Because a is in all equality groups that b is in, in all iterations of MARKACTIONS between i_a and i_b, *SubLength* and the length of the (in this case unique) *EqualActions*-set cannot differ, because otherwise a new equality group would emerge, containing b but not a. This results in node b *(*and all nodes between a and b*)* acquiring all equality tokens from node a via the lines 44-49 in MARKACTIONS, but not gaining a different unique token, as can be seen in the lines 39-43 in MARKACTIONS.

Case c): The nodes a and b are different kinds of nodes

The nodes a and b are different kinds of nodes: A parallel node and a choice node cannot have all their equality groups in common, so we only have to account for the case of an action node and a choice node (parallel node) being in the same equality groups. This case can only occur when the choice node (parallel node) is a (not necessarily directly) succeeding node of the action node. The argumentation is very similar to the one in case b2). Without loss of generality, let a be the action node and b be the choice node (parallel node), and let i_a and i_b be the iterations of MARKACTIONS in which their tokens are assigned, respectively. Because b is in all equality groups that a is in, in all iterations of MARKACTIONS between i_b and i_a, *SubLength* and the length of the (in this case unique) *EqualActions*-set cannot differ, because otherwise a new equality group would emerge, containing a but not b. This results in node a *(*and all nodes between b and a*)* acquiring all equality tokens from node b via the lines 44-49 in MARKACTIONS, but not gaining a different unique token, as can be seen in the lines 39-43 in MARKACTIONS.

Appendix B.3 Minimality (proof sketch)

The algorithm creates a simple merge as early as possible:

A set of flow nodes only has the same tokens, caused by the algorithm, if they belong to the same equality group. As shown in the proof of correctness, the algorithm finds all flow nodes that belong to the same equality group and marks them with tokens accordingly. This means, the result is minimal, if the algorithm finds not only a *correct* possible point to merge branches with nodes with the same unique tokens, but the *earliest one*.

MERGEBRANCHES is invoked with a set of flow nodes following a specific exclusive choice (cf. lines 3-5 of MERGEPATHS). It identifies the highest token *tok*, with which at least two outgoing branches of this exclusive choice are marked (cf. lines 3-20). As the tokens grow with the upwards traversal, this is the "largest" set of flow nodes (considering the length of the sequence of flow nodes) in the same equality group succeeding an exclusive choice. We thereby identify the earliest possible point in the process models, where at least two outgoing branches could be merged.

CREATEMERGENODE now traverses down each branch following the nodes in *MergeArray*, until it reaches the flow nodes with the previously identified maximal token *tok* as their highest token (lines 4-18) and then creates a simple merge before these flow nodes. To be precise, it inserts a *MergeNode* whose predecessors are the former predecessors (cf. line 21) of the flow nodes in *MergeArray* and whose successor is the first flow node within the identified equality group (cf. line 23). This leads to the fact that the first (earliest) possible merge position is found and thus the result is minimal.

As we recursively invoke MERGEBRANCHES again (cf. lines 32-34) if there are unmerged outgoing branches, we ensure to repeat this identification for each outgoing branch of the exclusive choice and thus we create a simple merge "as early as possible" for each outgoing branch. This leads to a minimal process model.

Appendix B.4 Computational complexity (sketch)

The main method of the algorithm is MARKACTIONS, which is invoked recursively. It identifies equality groups in one layer of the process model by traversing (line 4) the flow nodes in one layer and invokes itself again with the predecessors of the flow nodes within these equality groups (line 52 and 56). As every flow node could be preceded by only one other flow node, the method invokes itself n times in the maximum, whereas n is denoted as the maximum of the number of goal belief states as this is equal with the number of flow nodes in the last layer and the number of sequential flow nodes in the longest path of the planning graph.

In an average case, the algorithm should be much more efficient, as the number of flow nodes in one layer decreases when reaching exclusive choices.

In the case, that all actions preceding the goal belief states are equal, the MARKACTIONS primitive gets invoked one time for each layer as it identifies one overall equality group. Assuming, that the subsequent flow nodes stay equal until the initial belief state, the number of flow nodes is constant for each layer. Thus, the method invokes itself *n* times, whereas n is the number of layers in the planning graph. At the other end, if all *n* actions preceding the goal belief states differ, the MARKACTIONS primitive is invoked *n* times for the next layer. As then each subset only contains one action, the nested while loop is iterated only once for each invocation. Thus, the code within the while loop is executed *n* times. Let us assume that in each layer the algorithm finds two equality groups with equal size. Thus, MARKACTIONS gets invoked 2 times in each layer, while the length of the *Preceding* subset decreases to n/2 with each iteration. Thus, the maximum number of executions of the code within the while loop is constantly *n* in all cases.

Within the while loop we need to consider a switch, which implies to use the case with maximum complexity for the further calculations. The case "ChoiceNode" is the most complex, as the outgoing branches of an exclusive choice get merged when reaching it. The most complex part of this case is the invocation of MERGEPATHS, which further invokes MERGEBRANCHES for the outgoing branches. MERGEBRANCHES iterates in two nested for loops over the Array of outgoing branches which results in a computational complexity of $O(n^2)$ with *n* as the number of outgoing branches of the exclusive choice. Further, it invokes CREATEMERGENODE. This primitive traverses each outgoing branch of the exclusive choice that could be merged, too. Further, it traverses along the branches until the first flow nodes of the identified equality group. Additionally, we need to traverse the outgoing branches of a nested exclusive choice, if present. Thus, the overall computational complexity of this primitive is $O(n^3)$.

As the code within the while loop of MARKACTIONS is executed *n* times at most and the most complex case "ChoiceNode" has a computational complexity of $O(n^3)$, the complete algorithm has an overall complexity of $O(n^5)$.

Appendix C Verification properties of constructed planning graphs

Appendix C.1 Soundness of the resulting planning graphs

Following van der Aalst (1998), it needs to be proven, that (1) for each belief state of the planning graph that could be reached from the initial state ($bs \in Part_{BS}$) a sequence of actions or control flow structures exist that leads from the belief state to a goal state, that (2) the instance of the process terminates, if an edge, resulting in a goal state is traversed, and that (3) there are no "dead actions", which means, for each action at least one feasible path exists that contains this action.

When considering an initial planning graph, constructed by means of an automated planning approach, this initial planning graph fulfils (1) and (3) obviously, as automated planning approaches like Heinrich et al. (2011) construct correct (cf. (1)) and minimal (i.e., there are no actions that could not be executed, cf. (3)) planning graphs. As our approach does not construct any new action, (1) and (3) is furthermore guaranteed for the planning graph resulting after construction of simple merges in an automated manner.

As our approach constructs simple merges so that the goal belief state has exactly one incoming edge and no outgoing edges, (2) is also guaranteed for the planning graph that is constructed by means of our approach.

In sum: Our approach constructs sound planning graphs if for each belief state of the initial planning graph that could be reached from the initial state ($bs \in Part_{BS}$) a sequence of actions or control flow structures exist that leads from the belief state to a goal state (1) and the initial planning graph does not contain dead actions (3).

Appendix C.2 S-Coverability of the resulting planning graphs

In order to ensure that the resulting planning graphs are covered by S-components, we need to ensure that the constructed planning graphs are (1) well-formed and (2) free choice.

Ad (1): Assuming a well-formed planning graph as the starting point for our approach then our approach constructs a well-formed planning graph too. This is true, as the constructed simple merges do not increase the number of reachable states and as they do not influence liveness (i.e., no transitions, actions in our terms, are added and states are only unified but no new states are added) of the planning graph.

Ad (2): Considering Definition 1, the transition function $R: BS \times A \rightarrow 2^{BS}$ associates to each belief state $bs \in BS$ and to each action $a \in A$ the set $R(bs, a) \subseteq BS$ of next belief states. Thus, each action (related to transitions in Petri nets) is connected to exactly one preceding belief state and has exactly one input edge (arc). According to Verbeek *et al.* (2001), process models (they refer to workflows) are free choice if for every two actions, the preconditions are either disjoint or identical. In our context of automated process planning, this issue is addressed by means of a control flow structure called R-XOR. The control flow structure R-XOR is used to denote two or more feasible solutions, which are functional equivalents to represent a particular (sub)structure and behavior in a process model (for details, cf. Heinrich *et al.*, 2015, pp. 9-10). However, the control flow structure R-XOR is inferred at runtime, which means, the conditions to select exactly one outgoing path out of a set of outgoing actions are only given at runtime. This selection could be done, for instance, on non-functional properties that are unknown or undefined at planning time.

Appendix D Additional considerations

Within our paper, we considered the creation of pattern-compounds as they represent well-formed and sound block-structured fragments of a process model. Thus, process models consisting of such pattern-compounds are more readable and therefore understandable for laymen.

In some cases, it could be favorable to reduce the amount of duplicate actions while ignoring pattern-compounds as then, even more duplicates could be removed. To enable this, only minor adaptions have to be performed. Considering this requirement, it is no longer sufficient to only use the token of nested exclusive choices to identify mergeability. Instead, choice constructs of nested exclusive choices have to be considered within MERGEBRANCHES:

- Within the comparison (lines 3-19) in MERGEBRANCHES choice constructs of nested exclusive choices need to be compared in detail instead of just using the tokens of exclusive choices as a criterion for mergeability.
- The termination criterion of the while loop in line 4 of CREATEMERGENODE has to be adapted to consider equal actions in outgoing branches of nested exclusive choices.