Automated Planning of Process Models: The Construction of Parallel Splits and Synchronizations

Authors:

Heinrich, Bernd, Department of Management Information Systems, University of Regensburg, Universitätsstr. 31, 93053 Regensburg, Germany, Bernd.Heinrich@ur.de

Krause, Felix, Research Center Finance & Information Management, University of Augsburg, Universitätsstr. 12, 86159 Augsburg, Germany, Felix.Krause@fim-rc.de

Schiller, Alexander, Department of Management Information Systems, University of Regensburg, Universitätsstr. 31, 93053 Regensburg, Germany, Alexander.Schiller@ur.de

Citation: Bernd Heinrich, Felix Krause, Alexander Schiller, Automated Planning of Process Models: The Construction of Parallel Splits and Synchronizations, Decision Support Systems, Volume 125, October 2019, Paper 113096, ISSN 0167-9236, DOI 10.1016/j.dss.2019.113096 (https://www.sciencedirect.com/science/article/pii/S0167923619301253)

Automated Planning of Process Models: The Construction of Parallel Splits and Synchronizations

Abstract:

Efficient business processes play a major role in the success of companies. Business processes are captured and described by models that serve, for instance, as a starting point for implementing processes in a service-oriented way or for performance analysis. To support process modelers via methods and techniques (e.g., algorithms) in an automated manner, several research fields such as process mining and automated planning of process models have emerged. In particular, the aim of the latter research field is to enable the automated construction of process models using planning techniques. To this end, an automated construction of control flow patterns in process models is necessary. However, this task currently remains a widely unsolved issue for the central patterns parallel split and synchronization.

We introduce novel concepts, which, in contrast to existing approaches, allow the construction of complex parallelizations (e.g., nested parallelizations and parallelizations with an arbitrary length of path segments) and are able to identify the set of feasible parallelizations. Moreover, we propose an algorithm facilitating the automated construction of parallel splits and synchronizations in process models. Our approach is evaluated according to key properties such as completeness, correctness and computational complexity. Furthermore, both the practical applicability within several real-world processes of different companies in various contexts as well as the practical utility of our approach are verified. The presented research expands the boundaries of automated planning of process models, adds more analytical rigor to automatic techniques in the context of business process management and contributes to control flow pattern theory.

Keywords: business process modeling, automated planning of process models, control flow patterns, business process management

1 Introduction

The way a company defines and handles its business processes is of paramount importance for the company's success; this has been acknowledged in both science and practice over the previous years [1,2] and has started and stimulated research fields such as Business Process Management (BPM). A business process can be defined as the "specific ordering of work activities across time and space, with a beginning, an end, and clearly identified inputs and outputs" [3]. BPM focuses on capturing, implementing, analyzing and optimizing a company's business processes. In this regard, several research fields in BPM such as process mining [4–6], automated (web) service selection and composition [7,8] and automated planning of process models [9,10] have emerged in order to support business analysts and process modelers via methods and algorithms. In particular, the focus in this paper lies on the research field automated planning of process models, which aims to enable the automated construction of process models using planning algorithms [9–15].

The automated construction of process models can be understood as a planning problem [16] with the objective to arrange process model components in a feasible order based on an initial state, a set of available actions as well as conditions for goal states. The input data for this planning can, for instance, be obtained by fresh modeling of actions, extracting actions from existing process models or a conceptualization of (web) services to represent the corresponding actions [17]. Furthermore, interfaces of process modeling tools may be used (cf. *Evaluation*). A fundamental challenge of the automated construction of process models is to cope with control flow patterns describing the control flow of a process. More precisely, in order to plan sophisticated process models, not only a specific sequence of actions but also the control structures representing these patterns have to be constructed in an automated manner.

This general problem of planning an entire process model including control flow patterns is decomposed into subproblems to address a subproblem in-depth. Parallel splits (sometimes also called AND-splits) and their corresponding synchronizations capture elementary aspects of processes and thus are assessed to be central patterns [18–20]. Parallelizations are also deemed highly relevant when aiming to represent complex process flows (cf., e.g., examples in [21,22] and the discussion below). Furthermore, uncovering and representing the concurrent behavior of a system has long been assessed as valuable in many application contexts [23,24] and parallelizations are crucial, for instance, to reduce execution times of processes and service compositions [25]. Besides the relevance discussed by researchers, in several projects with different companies, we observed that almost all of the processes incorporated many parallelized actions. For example, in a cooperation with a European financial services provider in which over 600 core business processes were analyzed, over 90% of these processes contained at least one parallelization while around 33% contained more than five. Our analyses of these processes showed that the parallelizations served different reasons such as reducing total required execution time, increasing throughput and allowing a relatively constant workload of employees and a high utilization of resources (due to the reduction of waiting time). In this vein, parallelizations offer valuable decision support, as parallelizations enhance the decision-making aspect of process models [26,27]: They allow to select a beneficial way for process execution (e.g., in terms of execution time). Moreover, in some cases, they were necessary to ensure legal and regulatory compliance (e.g., to realize a dual principle). Furthermore, they improved organizational flexibility. For instance, they enabled a concurrent process execution by different organizational units and, due to reduced execution times, a quicker response to external events (e.g., customer complaints). This illustrates the practical importance of parallelizations in process models.

Addressing both the scientific and practical relevance, in this paper we will concentrate on the so far widely unsolved issue of an automated construction of parallel splits and synchronizations in process models. The contributions are as follows:

- Concepts are developed allowing the construction of complex parallelizations (including nested parallelizations and an arbitrary length of path segments within parallelizations) and the set of all feasible parallelizations while not constructing infeasible parallelizations. These concepts are independent of a concrete modeling language and can cope with possibly infinite sets of world states and large domains. This guarantees a maximum of compatibility with existing approaches and process modeling languages.
- Based on these concepts, we propose a novel algorithm for the automated construction of parallel splits and synchronizations in process models.
- The presented algorithm is implemented into a prototype which is evaluated in real-use situations.

The remainder of the paper is organized as follows: The next section contains the background of our research. Here, the theoretical background, the related work and the underlying planning domain are presented. Thereafter, we answer the key research question of how parallel splits and synchronizations can be constructed in an automated manner by proposing concepts and providing a concrete algorithm. The approach is illustrated by means of a running example. In the subsequent section, the concepts and the algorithm are evaluated according to key properties such as completeness, correctness and computational complexity. Furthermore, they are implemented into a prototype and their practical applicability within several real-world processes of different companies in various contexts as well as their practical utility are assessed. Finally, the last section summarizes the results, discusses limitations and provides an outlook for future research.

2 Background

In this section, we describe the theoretical background of our research based on the discussion by Soffer et al. [18] and present related work and the research gap. Thereafter, we outline the underlying planning domain.

2.1 Theoretical Background

Business process models are critical when designing, realizing and analyzing business processes [2,22,28,29]. Imperative models representing business processes usually consist of at least two types of components: actions and control flow patterns. These control flow patterns can be seen as a theory for clarifying the process flow, with a control flow pattern being a proposition which expresses how processes can be executed, or, more precisely, which control flows can exist in processes [20,30]. On the one hand, control flow patterns are abstract concepts striving to show the process flow independently of a concrete modeling language; on the other hand, modeling languages provide a concrete representation for control flow patterns [20]. The basic control flow patterns are sequence, exclusive choice, simple merge, parallel split and synchronization [19,22,30,31]. Control flow patterns allow to abstract from an individual process execution: In this regard, a parallel split specifies that a single route of execution is split into two or more sequences of actions (called ‘path segments’), where all actions in these different path segments can be executed concurrently [19,22,30]. However, the actions in different path segments originating from a parallel split do not necessarily have to be executed in parallel from a temporal perspective [19], although it is generally feasible to do so. Further, a synchronization represents a point where two or more path segments of arbitrary length originating from previous parallel splits converge into a single subsequent path [22]. This conceptualization regarding parallel splits and synchronizations also holds for so called nested parallelizations. Such a nested parallelization occurs when one or more parallelizations and their corresponding actions are contained in a path segment of another parallelization.

To further substantiate this conceptualization, the process state (denoted by its state variables; cf. Definition 1 in *Planning Domain*) has to be considered. In this way, potential inconsistencies can be avoided, ensuring the feasibility of parallel splits, synchronizations and their state transitions (cf., e.g., [32]). The well-known ACID properties [33] serve as reference to address this feasibility. More precisely, a synchronization merging two or more path segments (originating from a previous parallel split) requires that all actions in these path segments have been executed [22], while conflicts have to be avoided. For instance, when the same state variables are changed concurrently in different path segments, this represents a violation to the ACID-principle isolation, thus creating a conflict when trying to synchronize the path segments and their resulting states. In detail, while due to the potential concurrency of path segments leading to a synchronization, different actual execution routes are enabled (e.g., due to different possible temporal orders of actions), all of these routes need to result in the same state when synchronized. This holds due to two reasons: First, the state before the parallel split is equal. Second, it is necessary to be able to continue with the process independently of the actual execution route taken before synchronization [18]. Furthermore, as processes may be executed many times with different initial states, both control flow patterns as well as states (and its state variables) denoted by a process model should be able to deal with possibly infinite sets of world states and large domains as well as respective data types used by the state variables [12].

Based on these theoretical considerations with regard to control flow patterns, and in particular parallelizations, much work has been carried out to analyze control flow patterns in terms of different aspects such as inclusion in workflow modeling languages and corresponding tools (e.g., [22]), reconstruction of control flow in processes via process mining (e.g., [34]), empirical evidence and applications in real-world processes (e.g., [30]), and automated verification of control flow (patterns) (e.g., [35]). In the same vein, approaches for the automated planning of process models can also be seen as contribution to control flow pattern theory by analyzing and evaluating whether control flow patterns can be constructed correctly in an automated manner. Based upon this, sequences of actions as well as control flow patterns can be constructed in order to plan sophisticated process models. To this end, concepts and algorithms for the automated construction of control flow patterns need to be provided. In this paper, we contribute to this research by presenting concepts and an algorithm that constructs both parallel splits and synchronizations in an automated manner while considering the theoretical conceptualization of parallelizations discussed above.

2.2 Related Work and Research Gap

We structure existing approaches for the automated identification or construction of parallelizations according to the BPM lifecycle phases process modeling, process implementation, process execution and process analysis [36]. While our research focuses on the *process modeling* phase, we have also included relevant approaches from other phases, as such approaches may possibly be interesting.

In the *process modeling* phase, so far only the approach of Hoffmann et al. [10] discusses the automated construction of process models including parallelizations. However, the authors do not aim to provide concepts of how to construct parallelizations and do not present a concrete algorithm for the construction of parallelizations. Moreover, they use a heuristic approach in model-based software development, and thus their approach does not provide all feasible parallelizations.

Automated web service composition can be seen as part of the phases *process implementation* and *process execution* and is partly based on planning techniques [37–39]. Heinrich et al. [40] analyze multiple approaches [41–48] in detail regarding the construction of control flow patterns: Focusing on parallel splits and synchronizations, most of these approaches state that two actions can be parallelized if they do not contradict each other. However, these approaches do not define concepts and thus do not specify when exactly an action is contradicting another action. This would be necessary to provide a concrete automated planning algorithm for the construction of parallelizations. Only Meyer and Weske [43] state a formal concept to parallelize two actions, which is based on preconditions and effects not being in conflict. However, using this approach and focusing on two actions means that the length of each path segment within parallelizations is limited to only one action (cf. [43]). Moreover, construction of complex parallelizations such as nested parallelizations is not supported. Additionally, due to its heuristic nature, the authors do not aim to provide the set of feasible parallelizations. Furthermore, large sets of world states and large domains as well as respective numerical data types and also other large data types of state variables are not treated.

Other authors in these phases propose to calculate so called dependency coefficients for each action and suggest to parallelize two actions if their dependency coefficients are the same [49–52]. Dependency coefficients represent how many actions are dependent on the considered action or how many actions the considered action is dependent on. However, similarly to [43], the parallelized path segments are synchronized in any case after at most one action per path segment. Furthermore, nested parallelizations are not supported, and the approaches are heuristic. Additionally, large sets of world states as well as respective numerical data types and other large data types of state variables are not treated. The same holds for a similar approach proposed by Madhusudan and Uttamsingh [53] which divides a sequence of actions into sets of actions that can be parallelized based on precedence constraints.

Further research related to our work is associated with the phase *process analysis*. In process mining, data about executed processes is stored in logs and used to enable the *re*construction of process models. For instance, Hwang and Yang [54] present an approach in which process log data can be used to reconstruct the underlying process model and thus also control flow patterns such as parallel splits. The reconstruction of parallel splits and synchronizations in this research field is based on the execution order of actions discovered in the logs. Most approaches state that two actions are parallel if they appear in any order (see, e.g., [55–57]). This is of heuristic nature and a non-sufficient criterion, as, for instance, two actions may be executed in any order but not in parallel and at the same time because the same executing person (resource) is required for both actions. Other approaches also use logs with explicit timestamps enabling the identification of actions which were actually executed simultaneously [56,58] or detecting overlapping actions [34]. However, their intention and the presented algorithms are different to our research goal, since process mining focuses on the *re*construction of models for already *existing* processes. Therefore, these works do not aim to provide an approach for an automated construction of parallelizations in newly planned process models and thus do not present concepts to support this task. Moreover, as they rely on logs from existing process executions, these works do not deal with infinite sets of world states and large domains as well as respective data types used by the state variables. Further, Jin et al. [59] propose an approach for refactoring process models and including parallelizations in the refactored process models. They do so by applying techniques from process mining and determining relations between actions, allowing to identify actions which can be parallelized. However, the authors strive to refactor *existing* process models and thus do not aim to construct parallelizations in newly planned process models. Additionally – as Jin et al. [59] state – their approach cannot guarantee that the resulting process models are sound structured, which makes the manual intervention of a modeler necessary when applying the approach. This impedes an automated construction of parallelizations by means of an algorithm. Furthermore, the presented approach strictly relies on petri nets and is thus dependent on a concrete modeling language.

To sum up: In the literature there are several valuable contributions regarding an automated identification or construction of parallel splits and synchronizations which could serve as a basis for our research. However, there is a research gap which can be stated in terms of the following relevant aspects (cf. Section *Theoretical Background*) not addressed by existing approaches (cf. Table 1):

(A1) Concepts stating how to construct feasible parallelizations in newly planned process models need to be provided. These concepts have to allow the construction of complex parallelizations, which means, the support of nested parallelizations and an arbitrary length of path segments within parallelizations. The concepts must ensure the consistency of the state transitions resulting from a parallelization and must be formally and clearly defined.

(A2) Possibly infinite sets of world states and large domains as well as respective large data types of state variables have to be treated.

(A3) The set of feasible parallelizations has to be provided while preventing infeasible parallelizations.

(A4) The approach needs to be independent of a concrete modeling language.

(A5) A concrete algorithm for an automated construction of parallelizations in newly planned process models has to be provided.

Table 1. Overview of related work

Phase	Works	(A1)	(A2)	(A3)	(A4)	(A5)
Process Modeling	Hoffman et al. [10]	✕	✕	○	✓	✕
Process Implementation & Process Execution	Binder et al. [41], Constantinescu et al. [42], Pathak et al. [44], Bertoli et al. [45], Bertoli et al. [46], Pistore et al. [47], Lécué et al. [48]	✕	○	✕	○	✕
	Meyer & Weske [43]	○	✕	○	✓	○
	Omer & Schill [49], Omer [50], Rathore & Suman [51], Vanitha et al. [52], Madhusudan & Uttamsingh [53]	○	✕	○	✓	○
Process Analysis	van der Aalst et al. [55], Wen et al. [56], van der Aalst [57], Weijters et al. [58], Wen et al. [34]	✕	○	○	○	✕
	Jin et al. [59]	✕	✕	○	✕	✕

✓: considered; ✕: not considered; ○: partly considered

2.3 Planning Domain

Based on control flow pattern theory, when planning process models, we have to cope with an abstraction from individual process executions. Therefore, the realizations of state variable values are not determined at the moment of planning and belief states instead of world states need to be considered [16]. Here, a belief state represents possibly infinite sets of world states. When working with belief states it is common to deal with a nondeterministic planning problem and to refer to a nondeterministic planning domain. Both guarantee a maximum of compatibility with existing approaches in the literature [12–14,37,46,60] and allow an acceptance and use of our approach. Central for the nondeterministic planning domain is the nondeterministic belief state-transition system. It is based on the notion of a belief state tuple, which is defined as follows:

Definition 1 (*belief state tuple*). A *belief state tuple p* is a tuple consisting of a *belief state variable v(p)* and a subset *r(p)* of its predefined *domain dom(p)*, which is written as *p:=(v(p),dom(p),r(p))*. The domain, *dom(p)*, specifies which values can generally be assigned to *v(p)*. The set *r(p)⊆dom(p)* is called the *restriction* of *v(p)* and contains the values that can be assigned to *v(p)* in this specific belief state tuple *p*.

According to this definition, each belief state variable *v(p)* has a predefined data type (for example 'double') specifying the predefined domain *dom(p)*. Additionally, restrictions *r(p)* can be defined for each belief state variable *v(p)*. A restriction can either be described by logical expressions defining a set of values or an explicit enumeration of values. The notion of a belief state tuple is used in the formal definition of a nondeterministic belief-state transition system presented in the following. It is given in terms of its belief states, its actions and a transition function which describes how the application of actions leads from one belief state to possibly many belief states [16,46,61].

Definition 2 (*nondeterministic belief state-transition system*). Let BST be a finite set of belief state tuples. A *nondeterministic belief state-transition system* is a tuple $\Sigma = (BS, A, R)$, where

- $BS \subseteq 2^{BST}$ is a finite set of *belief states*. An element of BS, a belief state, is a subset of the finite set of belief state tuples BST, containing every belief state variable one time at the most.
- 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. The term *preconditions* (including inputs) denotes everything an action needs to be applied, including tangible and non-tangible entities (e.g., data, materials, components), general conditions (e.g., time slot when an action is applicable) and resources (e.g., staff, machines). The term *effects* (including outputs) denotes everything an action provides, deallocates or alters after it was applied, including tangible and non-tangible entities, general conditions and resources.[1]

– An action a is *applicable* in a belief state bs iff $\forall w \in precond(a)\ \exists u \in bs$: $v(w)=v(u) \wedge r(w) \cap r(u) \neq \emptyset$. In other words, a is applicable in bs iff all belief state variables in $precond(a)$ also exist in bs and the respective restrictions of the belief state variables intersect.

– $R: BS \times A \longrightarrow 2^{BS}$ 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 Definition 2, a state variable of the preconditions and effects is defined as belief state tuple that consists of the name of the state variable, its domain and a set of values, all of which can be assigned to the state variable in a specific world state (according to an individual process execution). From a process modeling perspective, this is a natural way to express certain preconditions and effects of actions and allows to represent possibly infinite sets of world states.

Definition 3 ((*non-*)*determinism in state space*). An action a is *deterministic* in a belief state bs iff $|R(bs, a)| = 1$. It is *nondeterministic* if $|R(bs, a)| > 1$. If a is applicable in bs, then $R(bs, a)$ is the set of belief states that can be reached from bs by applying a.

Based on both Definitions 2 and 3, a planning graph can be generated by means of several existing algorithms that progress from an initial belief state to goal belief states (see for example [13,37,46,60]). Here, a planning graph is defined as:

Definition 4 (*planning graph*). A *planning graph* is an acyclic, bipartite, directed graph $G = (N, E)$ with the set of

[1] To give an example: With the help of preconditions, data entities such as securities order data entities as well as bank employees (human resources) can be specified which are needed to apply an action "process buying order". Its effects specify, for example, that the securities order data entities are altered and the previously allocated bank employees are deallocated.

nodes N and the set of edges E. Henceforth, the set of nodes N consists of two partitions: The set of action nodes $Part_A$ and the set of belief state nodes $Part_{BS}$. Each node $bs \in Part_{BS}$ represents one distinct belief state from the set BS of belief states in the planning graph. Each node $a \in Part_A$ represents an action from the set A of actions in the planning graph. The planning graph starts with one explicit initial belief state $bs_{init} \in BS$ and ends with one to possibly many goal belief states $bs_{goal_j} \in BS$.

Given Definition 4, a planning graph may consist of one to many paths. Here, a path is defined as:

Definition 5 (*path*). A *path* in a planning graph is a sequence (bs_{init},a_1,bs_2,a_2,...,a_n,bs_{n+1}) of belief state nodes and action nodes starting with the initial belief state and ending in exactly one goal belief state with each action being represented one time at the most.

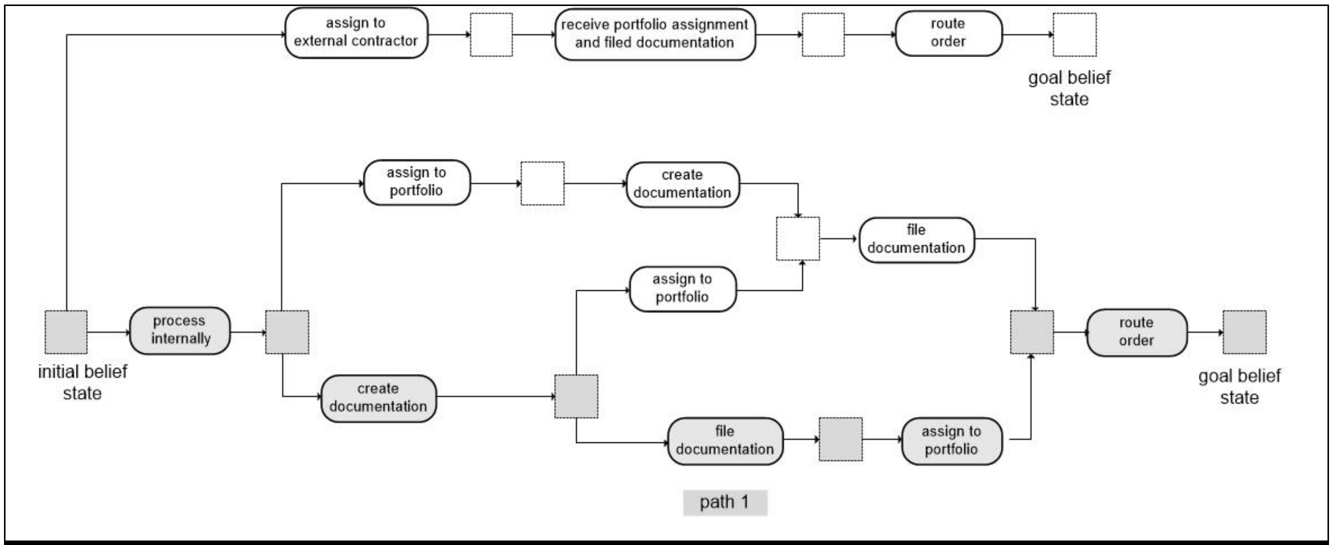


Figure 1. Excerpt of the order management of a financial services provider

To illustrate the above definitions of a planning domain and to introduce a running example, Figure 1 shows an excerpt of the real-world order management of a financial services provider. Here, the (internal or external) processing of an incoming order is performed. The full planning graph from which this example is taken can be found in the *Evaluation*. In our case the graph is planned by applying the approach suggested by Bertoli et al. [46]; however other approaches such as [61] are also feasible and provide the same graph. If a manually constructed graph (respectively, process model) is available, our approach may be applied as well to allow the construction of (additional) parallelizations for such models. The specification of the initial belief state and the condition for a belief state to be a goal belief state are given in Table 2.

Table 2. Initial belief state and condition for goal belief state	
initial belief state	{(order state, state, {passed}), (order price, double+, double+), (order amount, int+, int+), (internal processing, state, {unknown}), (documentation state, state, {not created}), (portfolio assignment, boolean, {false})}
condition for goal belief state	{(order state, state, {routed})}

In the initial belief state, an order has already been placed in terms of an order state, a price and an amount. The condition for a belief state to be a goal belief state of the presented excerpt represents that the order has been routed. Several actions are necessary before an order can be routed. The company can decide to mandate an external contractor (*assign to external contractor*) that provides a package which encapsulates all needed actions (*receive portfolio assignment and filed documentation*). After running these actions, the order can be routed (*route order*) to reach a goal belief state. If the company chooses not to mandate the external contractor, the action *process internally* enables the execution of three tasks which have to be completed before the order can be routed: *assign to portfolio*, *create documentation* and *file documentation*. The planning graph exhibits four possible sequences of actions to reach a goal belief state starting from the initial belief state and thus contains four paths (cf. Definition 5). In the following Table 3, we present the actions of one of the paths (marked in grey as path 1 in Figure 1) according to Definition 2. The remaining paths and actions are analogously annotated.

Table 3. Order management: Annotation of the actions of path 1

Action	Preconditions	Effects
process internally	{(internal processing, state, {unknown})}	{(internal processing, state, {true})}
create documentation	{(internal processing, state, {true}), (documentation state, state, {not created})}	{(documentation state, state, {created})}
file documentation	{(internal processing, state, {true}), (documentation state, state, {created})}	{(documentation state, state, {filed})}
assign to portfolio	{(internal processing, state, {true}), (portfolio assignment, boolean, {false})}	{(portfolio assignment, boolean, {true})}
route order	{(order state, state, {passed}), (order price, double+, double+), (order amount, int+, int+), (portfolio assignment, boolean, {true}), (documentation state, state, {filed})}	{(order state, state, {routed})}

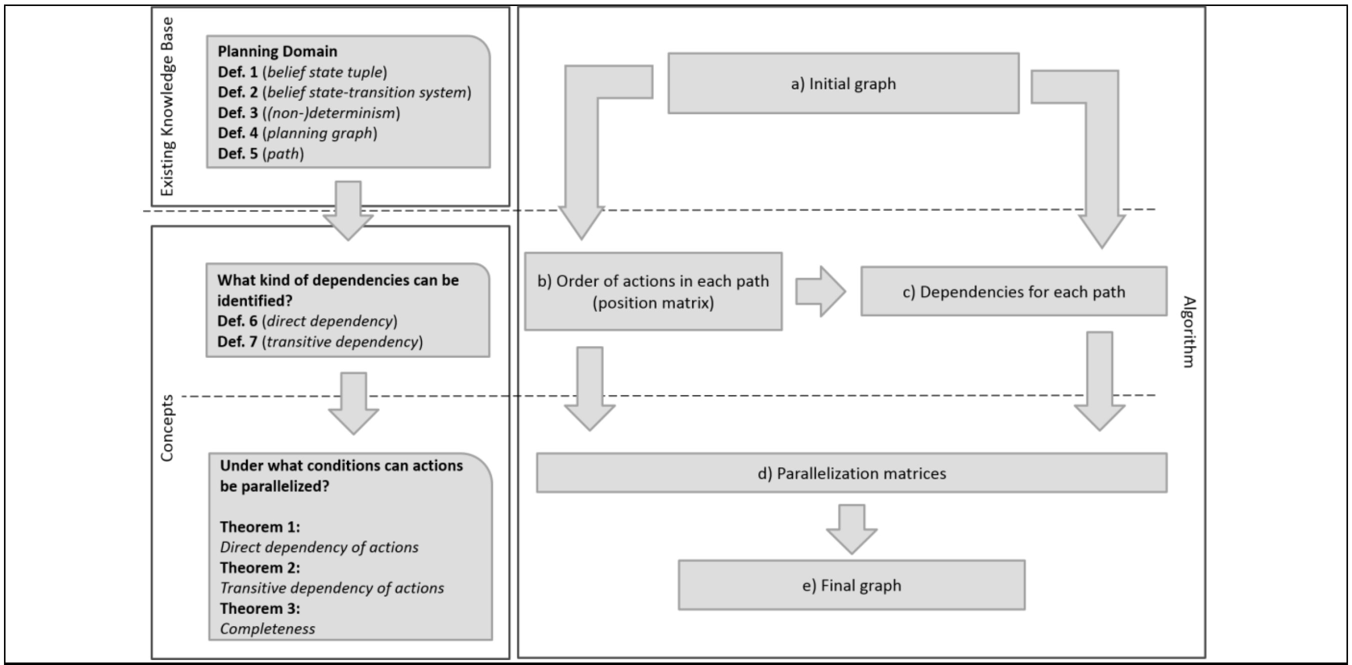
In path 1, the company chooses to process the order internally (action *process internally*), setting the value of the belief state variable `internal processing` to "true". Internal processing enables the creation of a documentation (action *create documentation*). This creation is represented by the belief state variable `documentation state` whose value is altered from "not created" to "created". After the documentation is created, it is filed. Therefore, the action *file documentation* requires the value "created" of `documentation state` and transforms it into "filed". Finally, the portfolio needs to be updated (action *assign to portfolio*), which alters the value of the belief state variable `portfolio assignment` to "true". Until now, the order could not be routed (action *route order*), since this requires a filed documentation as well as an existent portfolio assignment as represented by the preconditions of *route order*. Applying *route order* leads to the value of the belief state variable `order state` changing from "passed" to "routed". As this also represents the condition for a goal belief state, *route order is* the last action applied in the path.

3 Approach for the Automated Construction of Parallelizations

In this section, we present our concepts and algorithm for the automated construction of parallel splits and synchronizations. Figure 2 illustrates the approach on an abstract level by showing which part of the paper represents existing knowledge, which concepts we introduce and how the algorithm works.

We build our research on both the planning domain and planning graph (cf. Definitions 4 and 5; area a) in Figure 2), which can be constructed by existing algorithms. The graph contains all sequences of actions starting from the initial belief state and resulting in goal belief states. To provide a complete and correct solution to the problem of constructing the set of feasible parallelizations in a graph, we state concepts ("dependencies", cf. section *Concepts*) that describe conditions under which actions can be parallelized. To be more precise, we will first define "direct dependencies" between actions (cf. Definition 6). We will then show the connection of this notion to parallelizing actions. However, these direct dependencies will prove insufficient to construct the set of all feasible parallelizations, especially more complex parallelizations such as nested parallelizations. Therefore, we will introduce the concept of "transitive dependency" of actions (cf. Definition 7), critically complementing direct dependencies and enabling a correct and complete construction of parallelizations (cf. Theorems 1-3). More precisely, we will prove that if and only if neither of these dependencies occur, the regarded actions can indeed be parallelized.

An algorithm stating how to analyze these dependencies and how to construct all feasible parallelizations is described in the section *Algorithm*. For this analysis, it needs to be taken into account which action is succeeding another action in a certain path of the planning graph. To this end, our algorithm creates a position matrix representing the order of actions in each path of the planning graph (cf. area b) in Figure 2). Using this matrix and the identified dependencies (cf. area c) in Figure 2), parallelization matrices for each path of the planning graph can be constructed. These matrices show which actions are directly or transitively dependent on each other and which actions can be parallelized (cf. area d) in Figure 2) based on the respective path. When combined, the parallelization matrices therefore indicate every feasible parallelization and enable the construction of the final graph (cf. area e) in Figure 2) containing all parallelizations.


Existing Knowledge Base
Planning Domain
Def. 1 (belief state tuple)
Def. 2 (belief state-transition system)
Def. 3 ((non-)determinism)
Def. 4 (planning graph)
Def. 5 (path)
Concepts
What kind of dependencies can be identified?
Def. 6 (direct dependency)
Def. 7 (transitive dependency)
Under what conditions can actions be parallelized?
Theorem 1:
Direct dependency of actions
Theorem 2:
Transitive dependency of actions
Theorem 3:
Completeness
Algorithm
a) Initial graph
b) Order of actions in each path (position matrix)
c) Dependencies for each path
d) Parallelization matrices
e) Final graph
Figure 2. Overview of our approach for the construction of parallelizations

3.1 Concepts

The first idea to identify actions that can be parallelized is to compare the preconditions and effects of actions in a path. If this analysis shows that the effects of two compared actions are not disjoint from each other, or that the effects of one action intersect with the preconditions of the other action, we call this a direct dependency of both actions in the following.

Definition 6 (*direct dependency* ⇠): Let $(bs_{init},a_1,bs_2,a_2,...,a_n,bs_{n+1})$ be a path in the planning graph and let a_i and a_j be actions in this path with $i < j$ (i.e., a_j is succeeding a_i), $i \in \{1, \dots, n-1\}$, $j \in \{2, \dots, n\}$. The action a_j is *directly dependent* on the action a_i (denoted by $a_i \dashleftarrow a_j$) iff:

$$\left(v(effects(a_i)) \cap \left(v\left(precond(a_j)\right) \cup v\left(effects(a_j)\right)\right)\right) \cup (v\left(effects(a_j)\right) \cap v(precond(a_i))) \neq \emptyset$$

Here, $v(\dots)$ denotes the belief state variables of the tuples of the regarded set.

To illustrate Definition 6, consider the actions *process internally* and *create documentation* from the running example above. The effects of *process internally* and the preconditions of *create documentation* have the belief state variable `internal processing` in common. Therefore, these actions are directly dependent in every path containing both actions. This definition can be used to gain information about feasible parallelizations via the following theorem.

Theorem 1: Let (bs_{init},a_1,bs_2,a_2,...,a_n,bs_{n+1}) be a path in the planning graph and let a_i and a_j be actions in this path with $i < j$ (i.e., a_j is succeeding a_i), $i \in \{1, \dots, n-1\}$, $j \in \{2, \dots, n\}$.

a) If a_j is directly dependent on a_i (i.e., $a_i \dashleftarrow a_j$), a_i and a_j cannot be parallelized.

b) If a_j is *not* directly dependent on a_i and $j = i + 1$ (i.e., a_j is *directly* succeeding a_i), a_i and a_j can be parallelized.

Theorem 1 as well as all following theorems are proven in the *Supplement*. This theorem enables the construction of parallelizations with respect to directly adjacent actions. However, in order to construct complex parallelizations (including nested parallelizations and parallelizations with an arbitrary length of path segments), non-adjacent actions have to be analyzed as well. For that purpose, direct dependencies are not a sufficient concept, because it might or might not be correct to parallelize such actions that are not directly dependent. Therefore, we have to state under which additional concept it is feasible to parallelize two non-adjacent actions.

Definition 7 (*transitive dependency*): Let $p = $ (bs_{init},a_1,bs_2,a_2,...,a_n,bs_{n+1}) be a path in the planning graph and let a_i and a_j be actions in p with $i < j$ (i.e., a_j is succeeding a_i), $i \in \{1, \dots, n-2\}$, $j \in \{3, \dots, n\}$. The action a_j is *transitively dependent* on the action a_i in p iff there is a set $A_k = \{a_{k_1}, \dots, a_{k_m}\} \subseteq \{a_{i+1}, \dots, a_{j-1}\}$, $A_k \neq \emptyset$, such that $a_i \dashleftarrow a_{k_1} \dashleftarrow \dots \dashleftarrow a_{k_m} \dashleftarrow a_j$.

A transitive dependency in a path can be seen as a continuous chain of direct dependencies among a non-empty subset of actions in that path, leading from one action to another. Evidently, the concrete ordering of actions in a path plays a crucial role for transitive dependency: The actions $a_{k_1}, \dots, a_{k_m}$ that result in a transitive dependency of an action a_j on an action a_i in a path p might, even if they are contained in a path p', fail to do so in p' due to being in a different ordering (for example, in p', one of the actions $a_{k_1}, \dots, a_{k_m}$ may be executed after a_j). This underlines the need of a path-wise definition of transitive dependency. Definition 7 can be used to gain information about feasible parallelizations via the following theorem.

Theorem 2: Let $p =$(bs_{init},a_1,bs_2,a_2,...,a_n,bs_{n+1}) be a path in the planning graph and let a_i and a_j be actions in p with $i < j$ (i.e., a_j is succeeding a_i), $i \in \{1, \dots, n-2\}$, $j \in \{3, \dots, n\}$.

a) If a_j is transitively dependent on a_i, the actions a_i and a_j cannot be parallelized based on p.

b) If a_j is neither directly nor transitively dependent on a_i, the actions a_i and a_j can be parallelized.

Focusing only on a single path, we might at first “miss out” (from a graph-wise perspective) a certain parallelization by not parallelizing transitively dependent actions (cf. Theorem 2a)), if these actions are not dependent on each other in another path of the planning graph. However, the respective parallelization is then constructed based on the analysis of that path:

Theorem 3 (*completeness*): Let G be a planning graph consisting of the paths p_1,...,p_k. Suppose the actions $a_1, \dots, a_n$ represented in G can be parallelized. By analyzing direct and transitive dependencies in all paths p_1,...,p_k, the parallelization of a_1,...,a_n is constructed.

This result finalizes the development of our concepts. Thus, the set of feasible parallelizations including nested parallelizations and parallelizations consisting of path segments with more than one action can be constructed based on our formally defined concepts of direct dependency, transitive dependency and completeness.

3.2 Algorithm

In this section, we present an algorithm which builds on the concepts and allows to construct complete graphs while also being computationally efficient (cf. Section *Evaluation*). Let P be the set of all paths contained in the planning graph G (as planned by existing approaches; e.g., [46,61]). For each $p \in P$ we define a *parallelization matrix* M_p. The purpose of a parallelization matrix is to show which actions can be parallelized based on the respective path. To this end, our algorithm fills the parallelization matrices with entries determining whether to allow or to prohibit parallelization based on the concepts from the previous section. The family $\left(M_p\right)_{p \in P}$ then indicates all feasible parallelizations of the whole graph. The pseudo code of the algorithm is shown in the Table 4 (an extended version with comments is available in the *Supplement*). The algorithm relies on four steps, which are exemplified in the following by our running example:

Table 4. Pseudocode of our algorithm

```
1 Vector allActions:= new Vector()
2 [][] positionMatrix:= new int [#actionsInGraph][#pathsInGraph]
3 for all p ∈ (1 ≤ p ≤ #pathsInGraph)
4   for all i ∈ (1 ≤ i ≤ p.length)
5       if (a[i][p] ∉ allActions) then
6           allActions.add(a[i][p])
```

```
7        **end if**
8        positionMatrix[allActions.getIndex(a[i][p])][p] = i
9    **end for**
10 **end for**
11 Vector ParaMatrices:= new Vector()
12 **for all** $p \in (1 \le p \le \#pathsInGraph)$
13   [][]ParaMatrix:= new String[allActions.length][allActions.length]
14   ParaMatrices.insertElementAt(ParaMatrix, p)
15 **end for**
16 **for all** $p \in (1 \le p \le \#pathsInGraph)$
17   **for all** $i \in (2 \le i \le allActions.length)$
18       **if** (positionMatrix[i][p]=0) **then**
19           **continue**
20       **end if**
21       **for all** $j \in (i-1 \ge j \ge 1)$
22           **if** (positionMatrix[j][p]=0) **then**
23               **continue**
24           **end if**
25           **if** (ParaMatrices.elementAt(p).[i][j] $\neq$ ddep) **then**
26               **if**(v(effects(a[i])) $\cap$ (v(precond(a[j])) $\cup$ v(effects(a[j]))) $\neq \emptyset \vee$ v(precond(a[i])) $\cap$ v(effects(a[j])) $\neq \emptyset$) **then**
27                   **for all** $a \in (p \le a \le \#pathsInGraph)$ **do**
28                       **if** (positionMatrix[i][a]=0 $\vee$  positionMatrix[j][a]=0) **then**
29                           **continue**
30                       **end if**
31                       ParaMatrices.elementAt(a).[i][j] $\leftarrow$ ddep
32                   **end for**
33               **else**
34                   **if** (|positionMatrix[i][p]-positionMatrix[j][p]| = 1) **then**
35                       ParaMatrices.elementAt(p).[i][j] $\leftarrow$ para
36                   **end if**
37               **end if**
38           **end if**
39       **end for**
40   **end for**
41 **end for**
42 **for all** $p \in (1 \le p \le \#pathsInGraph)$
43   **for all** $i \in (3 \le i \le p.length)$
44       **for all** $j \in (i-2 \ge j \ge 1)$
45           pos_i:= allActions.getindex(a[i][p])
46           pos_j:= allActions.getindex(a[j][p])
47           **if**(ParaMatrices.elementAt(p).[Max(pos_i,pos_j)][Min(pos_i,pos_j)]) $\neq$ ddep) **then**
48               **for all** $k \in (i > k > j)$
49                   pos_k:= allActions.getindex(a[k][p])
50                   **if**((ParaMatrices.elementAt(p).[Max(pos_i,pos_k)][Min(pos_i,pos_k)]) = (ddep $\vee$ tdep))
51                   $\wedge$ (ParaMatrices.elementAt(p).[Max(pos_j,pos_k][Min(pos_j,pos_k)]) = (ddep $\vee$ tdep))) **then**
52                       (ParaMatrices.elementAt(p).[Max(pos_i,pos_j)][Min(pos_i,pos_j)]) $\leftarrow$tdep
53                       **break for**
54                   **end if**
55               **end for**
56           **end if**
57           **if**(ParaMatrices.elementAt(p).[Max(pos_i,pos_j][Min(pos_i,pos_j] $\neq$(ddep $\vee$ tdep)) **then**
58               ParaMatrices.elementAt(p).[Max(pos_i,pos_j][Min(pos_i,pos_j]$\leftarrow$para
59           **end if**
60       **end for**
61   **end for**
62 **end for**
```

1) A list of the actions in the graph and a position matrix, containing the position of each action in each path, is generated (line 1-10). To this end, first the actions of the graph are determined in the order in which they appear (line

3-7): this means, all actions of a first path (in our example, *process internally, create documentation, file documentation, assign to portfolio, route order*; cf. Figure 1) are followed by the actions in other paths that were not part of the first path (*assign to external contractor, receive portfolio assignment and filed documentation*)[2]. Then, the position matrix containing the position of every action in each path of G is generated (line 8). The rows represent the actions (in the order identified before), the columns correspond to the different paths. For our example with the four paths $p1$, $p2$, $p3$ and $p4$, this yields the following position matrix:

$$\begin{array}{c|cccc} & p1 & p2 & p3 & p4 \\ \hline pi & 1 & 1 & 1 & - \\ cd & 2 & 2 & 3 & - \\ fd & 3 & 4 & 4 & - \\ ap & 4 & 3 & 2 & - \\ ro & 5 & 5 & 5 & 3 \\ ae & - & - & - & 1 \\ re & - & - & - & 2 \end{array}.$$

Abbreviation	Action
pi	process internally
cd	create documentation
fd	file documentation
ap	assign to portfolio
ro	route order
ae	assign to external contractor
re	receive portfolio assignment and filed documentation

Here, " $-$ " denotes that the action is not part of the respective path.

2) A set of (at first, empty) parallelization matrices is constructed (lines 11-15). The rows and columns of every parallelization matrix M_p represent all actions contained in G ordered by their position as identified in step 1). Each entry determines a row-column-combination and therefore an action-action-combination. For our example, this means that four (one for each path) parallelization matrices M_1 to M_4 are generated, each row and column representing one of the seven actions contained in the graph.

3) The algorithm examines – for all paths – the direct dependencies between pairs of actions in the respective path (lines 16-41)[3]. Whenever a direct dependency is identified in a path p (line 26), it is inserted in the respective entry in M_p. The concept of direct dependency is path-overarching, so that additionally, to reduce computing time, an identified direct dependency is also inserted into all entries corresponding to these two actions in the subsequent paths (lines 27-32). Following Theorem 1a), direct dependencies prohibit parallelization. When actions are *not*

[2] A different order of the paths does not lead to different sets of feasible parallelizations.

[3] Only one entry for each pair of actions is required, so in this and the following steps, just a triangular matrix needs to be considered and without loss of generality, all entries above the main diagonal can be disregarded (cf. for-loops, e.g., line 21). Also, only matrix entries for actions that actually appear in the respective path need to be filled out (lines 18-24, lines 28-29).

directly dependent, it is examined whether one of the actions is directly succeeding the other action in the considered path (lines 33-37). This is done via the position matrix. If this is indeed the case, the potential parallelization is noted in the corresponding entry in M_p (line 35), which is justified by Theorem 1b). In our example, the analysis of the direct dependencies starts with the actions *process internally* and *create documentation*. The effects of *process internally* and the preconditions of *create documentation* have the belief state variable `internal processing` in common (cf. Table 3 for an overview of preconditions and effects), resulting in a direct dependency of those two actions. Therefore, this direct dependency is inserted in the parallelization matrices M_1 to M_3, since both actions are applied in the first three paths. The algorithm then examines *file documentation* and *create documentation* (directly dependent, because the effects of both actions contain the belief state variable `documentation state`), *file documentation* and *process internally* (directly dependent due to the common belief state variable `internal processing`), *assign to portfolio* and *file documentation* (not directly dependent because of no common belief state variable) etc. and inserts the respective entries in the parallelization matrices.

4) The transitive dependencies are worked out (necessarily path-wise; lines 42-62). Only actions which are not directly dependent and which are not directly succeeding each other remain to be examined, reducing computing time. The algorithm searches for a set A_k of actions as in the definition of transitive dependency (Definition 7). This is done in a special proceeding order to guarantee that all dependencies required to examine a certain transitive dependency have already been determined beforehand (cf. for-loops in lines 43, 44 and 48). More precisely, the algorithm at first searches for a transitive dependency by adjacent actions (for example between action 1 and action 3 by action 2). Thereafter, the algorithm searches for transitive dependencies between non-adjacent actions (so that, e.g., for examining the transitive dependency of action 4 on action 1, the potential transitive dependency between action 1 and action 3 can be already used). Every transitive dependency is noted in the parallelization matrix of the considered path, prohibiting parallelization (cf. Theorem 2a); line 52). If neither direct nor transitive dependency is discovered between two actions in a path, the actions can be parallelized (cf. Theorem 2b); lines 57-58). In our example, the first potential transitive dependency that the algorithm analyzes is the one between *create documentation* and *assign to portfolio* in path 1 (cf. Figure 1), since it is already known that *file documentation* is directly dependent on *process internally*. However, *assign to portfolio* is not dependent on *file documentation* (which

is the action applied in-between *assign to portfolio* and *create documentation*), and thus *create documentation* and *assign to portfolio* are not transitively dependent in path 1. Thus, the possibility to parallelize *assign to portfolio* and *create documentation* is included in M_1. The algorithm proceeds by examining the pair *route order* and *process internally* (in path 1). Here, *route order* is directly dependent on *assign to portfolio*, which itself is directly dependent on *process internally*, leading to a transitive dependency of *route order* and *process internally* that is entered in M_1. In this way, the algorithm identifies all transitive dependencies for all paths.

Based on the parallelization matrices, the process model containing all feasible parallelizations can be constructed by iterating over all parallelization matrices, including each feasible parallelization (as indicated in of the matrices) in the process model and removing redundant parallelizations. Thus, Theorem 3 is considered. For the running example, the completed parallelization matrices are

$$M_1 = M_2 = M_3 = \begin{array}{c|ccccccc} & pi & cd & fd & ap & ro & ae & re \\ \hline pi & - & - & - & - & - & - & - \\ cd & ddep & - & - & - & - & - & - \\ fd & ddep & ddep & - & - & - & - & - \\ ap & ddep & para & para & - & - & - & - \\ ro & tdep & ddep & ddep & ddep & - & - & - \\ ae & - & - & - & - & - & - & - \\ re & - & - & - & - & - & - & - \end{array} \qquad M_4 = \begin{array}{c|ccccccc} & pi & cd & fd & ap & ro & ae & re \\ \hline pi & - & - & - & - & - & - & - \\ cd & - & - & - & - & - & - & - \\ fd & - & - & - & - & - & - & - \\ ap & - & - & - & - & - & - & - \\ ro & - & - & - & - & - & - & - \\ ae & - & - & - & - & tdep & - & - \\ re & - & - & - & - & ddep & ddep & - \end{array}.$$

They are used to construct the final graph (depicted in Figure 3) including all feasible parallelizations.

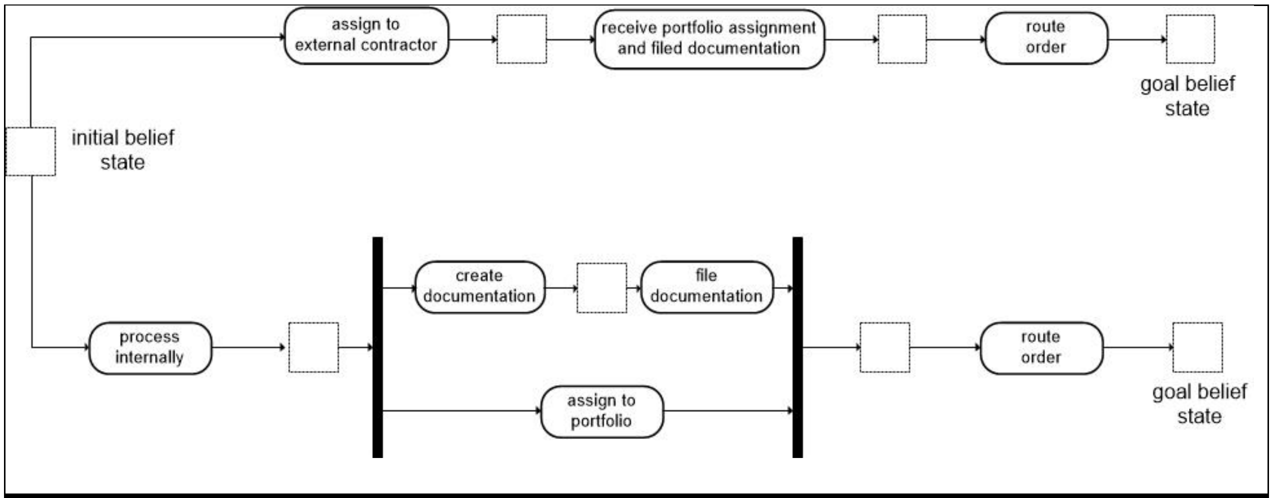


Figure 3. Final graph resulting from the application of the algorithm to the running example

4 Evaluation

The presented approach was evaluated as shown in this section.

4.1 Analysis of the algorithm properties

We mathematically evaluated the algorithm in terms of the key properties termination, completeness and computational complexity and summarize the results in the following (proofs and calculations are available in the *Supplement*).

Termination: The algorithm terminates.

Correctness/Completeness: The algorithm leads to complete and correct parallelization matrices: Every required entry is inserted and there is no entry that would allow an infeasible parallelization or prohibit a feasible parallelization.

Computational Complexity: When evaluating the computational complexity of our algorithm, we considered the worst-case-scenario as is usual. The following results were achieved: Given a planning graph in which each path has n actions and each action has m preconditions and m effects, the asymptotic time complexity of our algorithm is $O(n^3)$ and $O(m^2)$. This polynomial run time underlines the computational efficiency (cf. [62,63]) and thus practical applicability of the algorithm. We did not evaluate the computational complexity of our algorithm in comparison to competing algorithms since it solves a heretofore unsolved problem (cf. aspects (A1) – (A5) in *Related Work and Research Gap*).

4.2 Operational evaluation

To examine its technical feasibility and practical applicability [64], we examined our approach with respect to the following three evaluation questions:

(*E1*) Can the algorithm be realized in a prototypical implementation?

(*E2*) Can the algorithm be applied to real-world processes and how can the necessary input data (i.e., the specification of actions, initial belief state and conditions for goal belief states) be obtained?

(*E3*) Which output results from the application of the algorithm to real-world processes?

In regard to (*E1*), a Java implementation of an existing algorithm for the automated construction of planning graphs

[46] served as a basis for our work. This implementation allows the import of actions, initial belief states and conditions for goal belief states specified in form of XML files. We extended the implementation to incorporate the presented algorithm for the automated construction of parallelizations. The validity of the prototype was ensured by means of structured tests using the JUnit framework and planning test process models. At the end of the test phase, the implementation did not exhibit any errors. This result supports the technical feasibility of the algorithm and provides "proof by construction" [65,66] [4].

With respect to (*E2*) we analyzed the algorithm in-depth in different real-use situations using our prototypical implementation[5]. In the following, we exemplarily focus on one of these real-world processes referring to the order management of a European financial services provider (the running example used above is part of this process as well). More precisely, this process addresses the execution of security orders where several steps including check routines have to be modeled (cf. Figure 4). In the past, this process had to be (re)designed several times due to new services, new regulations or changing organizational requirements (for example, when outsourcing parts of the process to external service providers). To evaluate our approach we focused on the previous redesigns of this process and analyzed whether it is possible to apply the approach in these redesign situations and to what extent the results of the automated planning match with manually designed parallelizations.

In order to apply the algorithm, we conducted two steps: First, we obtained the necessary input data. To do so, a set of actions was extracted based on former process models in the area of security order management. This could be done easily and in an automated manner via the financial services provider's process modeling tool (ARIS toolset) which features a XML interface. Such an interface can be used in order to export actions to our prototype. In the area of security order management, about 200 different actions including their preconditions and effects were imported from the ARIS toolset and verified. Besides, a small number of additional actions was modeled manually. Moreover, the initial belief state and conditions for goal belief states were specified in cooperation with the financial

[4] In this context, a web interface for the implementation capable of planning process models in an automated manner has been prepared. It can be accessed using the following link: http://www-sempa.ur.de/

[5] The prototype was run on an Intel Core i7-2600 3.40 GHz running Windows 7, 64 Bit and Java 8, Build-Version 1.8.0_05-b13.

services provider. Then, the process models were planned using the prototype. This second step took less than two seconds in case of the order management process model.

Concerning (*E3*), we examined the output. Figure 4 shows an entire planned process model[6]. Here, our algorithm constructed two parallelizations which were also part of the manually designed process model. The first parallelization is constructed after the action *proof stock*, where the actions *enter quantity* and *determine market value* are parallelized. The second parallelization refers to our running example. Here, the action *assign to portfolio* is parallelized to the actions *create documentation* and *file documentation*. The assessment underlined the applicability and feasibility of the algorithm in all redesign situations of the security order management process.

To further address the evaluation questions, the presented approach was applied in additional real-use situations from various application contexts and different companies. These applications are discussed in the *Supplement*. The analysis of the evaluation questions (*E1*)-(*E3*) supported the technical feasibility and practical applicability of the presented approach. Table 5 summarizes the results.

Table 5. Results with regard to the evaluation questions (*E1*)-(*E3*)

Evaluation Question	Result
(*E1*) Can the algorithm be realized in a prototypical implementation?	The algorithm was implemented and successfully integrated into a prototype for the automated planning of process models.
(*E2*) Can the algorithm be applied to real-world processes and how can the necessary input data (i.e., the specification of actions, initial belief state and conditions for goal belief states) be obtained?	The algorithm was applied in several real-use situations of various application contexts and different companies. The analyzed situations included up to 278 actions and 189 states in the planning graph and are of a medium to large size. This is also reflected in the number of paths of the different planning graphs which ranges up to over 1.2 million (due to the various orders the actions can appear in). The necessary input data could, for example, be obtained by the XML interface of an existing modeling tool. Our algorithm was able to cope with the required data types and could be applied in all situations without restrictions. The run time of the algorithm varied – depending on the size and complexity of the processes - from a few milliseconds up to around 12.5 minutes.
(*E3*) Which output results from the application of the algorithm to real-world processes?	The algorithm constructed parallelizations for each of the real-world processes. For a significant number of processes, complex parallelizations (e.g., nested parallelizations) were constructed. The algorithm provided the manually constructed parallelizations and further, additional feasible parallelizations.

[6] Note that in this figure, the two paths of our running example have been merged before the action *route order*, since the process model is represented as a UML activity diagram without state nodes. This diagram type was the modelling notation preferred by the financial services provider.

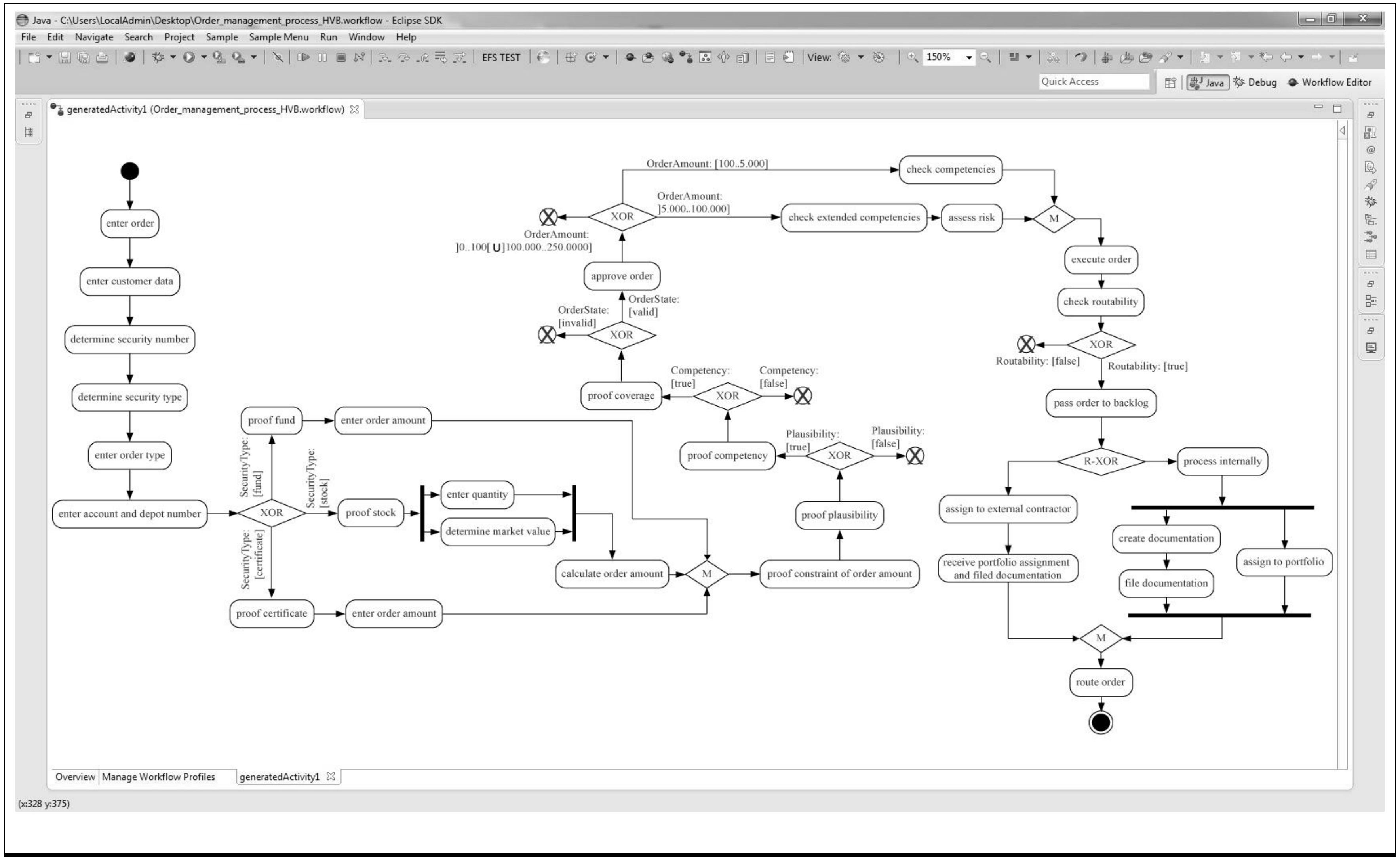


Figure 4. Planned model of the order management process (screenshot prototype)

4.3 Practical utility

We further assessed the practical utility [64] of our approach by means of a naturalistic ex post evaluation [67]. Its application resulted in the construction of the parallelizations already contained in the (existing) manually designed process models as well as additional feasible parallelizations and consequently increased flexibility by definition (cf. [28]). Thereby, flexibility by definition represents the ability to consider alternative execution routes at planning time (in our context, facilitated by feasible parallelizations). This capability is of practical use for decision support because alternative execution routes can be assessed based on economic and resource criteria constraints. Subsequently the most beneficial execution route can be selected for process execution. For instance, in this way, an execution route with favorable execution time may be chosen when necessary. The real-use situation of this naturalistic evaluation is presented in the following Table 6 [67,68].

Table 6. Real environment analyzed in the naturalistic evaluation

General setting	Extensive project at a European financial services provider aiming for an improved transparency of costs, execution times and capacities with regard to core business processes
Available data and systems	Detailed information as well as key economic indicators such as total cost, total required execution time and personnel requirements for a large number of business processes and the actions covered by these processes; provided by process experts and executives of the financial services provider
Involved people	Multiple organizational units of the European financial services provider and their employees (business and process experts, executives)
Hypothesis	Realizing a previously non-identified feasible parallelization should reduce total costs and total required execution times while increasing resource utilization, as long as the necessary resources for concurrent execution are available. This should also help in the prevention of errors and claims occurring during process execution.

Similar to Siha and Saad [69], we exemplarily discuss two selected cases in the context of the "Contracting wealth management customer" process (cf. Table C.1 in *Supplement*) in Table 7.

Table 7. Selected cases in the naturalistic evaluation

Subprocess	**Managing depot conditions**	**Handling non-executed security paper orders**
Description of the sub-process	Customer inquiries lead to changed depot conditions which are issued by the respective employees in charge. These change requests are stored in a list, which has to be worked through by different organizational units of the financial services provider to complete the needed change.	A variety of problems results in non-executed security paper orders issued by employees in charge of the financial services provider. These orders need to be rectified, forwarded and executed.
Organizational units involved	Advisors / multiple regional service divisions / processing department / process management department	Advisors / regional service division / commerce, sales and deposits units / processing department / process management department / financial market services
Issue	The previously existing sequential execution of actions occurring when, for instance, a customer opened a deposit account had resulted in a significant time gap between the opening and the completion of the respective inquiry. This, in turn, had led to customer complaints and repeated effort of the employees in charge.	Discussions with different organizational units revealed that for certain actions, it had not been clear which unit was in charge. Time delays resulting from the sequential execution of these actions had resulted in long execution times and many unnecessary internal inquiries and reworks. This in turn had led to claims of customers because overdue security paper orders had been deleted erroneously.
Improvement potential	A clear division of responsibility between the different organizational units of the financial	Our analysis showed that, as long as different organizational units were responsible for some of the actions, a

	services provider allowed a (previously not identified) concurrent execution of actions (i.e., nested parallelizations). The feasibility of this concurrent execution of actions with respect to economic criteria and resource constraints was confirmed by experts in a workshop based on which the employees in charge were informed and trained.	parallelization of these actions was not only feasible, but highly beneficial. A workshop with the respective organizational units (including, e.g., the sales, commerce and deposits units) was conducted to ensure that the proposed concurrent action execution would be possible based on economic criteria and resource constraints. Thereby, it was also ensured that each organizational unit was only in charge of the actions it was capable for.
Results	The concurrent execution of previously sequentially executed actions could be realized. In this way, a large number of time delays and repeated efforts could be avoided. A 50%-reduction in occurrence of these aspects led to saving 20% of total required execution time. For the employees, this amounted to an average reduction of at least 12 minutes of working time per process execution. Additionally, realizing the improved feasible execution route including the concurrent execution of actions resulted in an optimization potential for cost savings of 1.2 full time equivalents p.a.	The concurrent execution of actions allowed an improved workload efficiency and thus an optimization potential for cost savings amounting to 1.42 full time equivalents p.a. Furthermore, due to a reduction of the total required execution time, the aforementioned claims could be reduced or even avoided.

Overall, our approach demonstrated its practical utility in the analyzed real-use situations with respect to the criterion flexibility by definition. Several in-depth analyses and discussions with executives and employees supported that realizing the identified concurrent execution of actions (e.g., in nested parallelizations) was feasible and beneficial based on economic criteria and resource constraints. After workshops with the involved organizational units of the financial services provider, selected execution routes including the concurrent execution were applied. In this way, total required execution times were reduced, resource utilization was increased and errors and claims could be reduced. In these real-use situations, an improved decision support provided by our approach was realized.

5 Conclusion, Limitations and Further Research

In this paper, we introduced concepts stating how to construct parallel splits and synchronizations in newly planned process models in an automated manner. Compared to existing works, our approach supports the construction of all feasible parallelizations in a process model, including complex parallelizations such as nested parallelizations. Based on our formally defined concepts, we presented a concrete algorithm for this task. We implemented the approach into a software prototype to show its applicability. Moreover, the presented approach allows the consideration of large data types and planning independently of a concrete modeling language. This means that applicability for various notations such as UML activity diagrams, BPMN diagrams and Event-driven Process Chains is supported.

The main findings from our research for control flow pattern theory are as follows. To begin with, the presented concepts support the foundations of control flow pattern theory regarding the patterns parallel split and synchronization and allow to show that both patterns can indeed be constructed feasibly and in an automated manner. Second, the theoretical understanding of parallel splits and synchronizations was furthered, compared to existing approaches: Thereby, interestingly, it was proven that for two or more actions to be parallelized, other actions have to be analyzed as well (due to potential transitive dependency). Third, we showed that actions which are directly or transitively dependent cannot be parallelized. This adds rigor to statements prevalent in literature that actions may not be "in conflict" or similar descriptions (cf., e.g., [70]). Fourth, it was proven that in contrast to existing concepts (e.g., based on the order of actions), the absence of dependency is indeed a sufficient criterion for actions to be parallelized.

Building on these insights, our work offers major findings for the research field automated planning of process models. We believe that by addressing the presented research gap, it significantly expands the boundaries of the research field. In particular, the proposed concrete algorithm for an automated construction of all feasible parallelizations in newly planned process models forms an indispensable component of a comprehensive approach for an automated planning of process models.

Additionally, there are implications for applying our approach in practice as well. Parallelizations are, amongst other purposes, used to reduce execution times and costs while increasing workload efficiency and resource utilization. This optimization potential can be leveraged by applying our approach which allows the construction of additional parallelizations, thus increasing flexibility by definition. In this way, our approach provides valuable decision support. To reflect such implications in more detail: First, proposing alternative feasible parallelizations opens the door for discussions with process managers and executives as specific and detailed models are on the table, which can be explored and assessed regarding their organizational feasibility. Second, such discussions and what-if scenarios are in particular very fruitful – as the experiences in our cooperations show – in cases where existing process models have to be adapted to new company-internal or external (e.g., new regulations) requirements. Third, because the run times to plan models were short, some preconditions and effects of actions, especially the ones which specify resources and organizational responsibilities, could be altered. In this way, new ways and alternatives to overcome traditional organizational constraints could be provided. Fourth, when process models are realized by (web) services, our approach can provide valuable input. For instance, the process models constructed by our approach can be used by service selection approaches. This means, planned process models including different feasible parallelizations can be assessed regarding both their potential service implementation and resulting Quality-of-service values (e.g., overall cost or availability) which supports to choose beneficial execution routes (cf., e.g., [17,71]).

However, our research also possesses some limitations that should be addressed in future work. First, our approach constructs parallelizations for planning graphs without cycles (cf. Definitions 4 and 5). This limitation could be resolved by analyzing the (sub)paths within a cycle once and separately, allowing the construction of parallelizations while considering arbitrary cycles. Further advanced control flow patterns and their combination with parallelizations have to be examined in a similar way. Second, when applying the approach in real-use situations, noisy preconditions or effects may occur and influence dependencies between actions. To address this issue, multiple plannings with different preconditions and/or effects of respective actions can be initiated. Based on this, it can be evaluated whether the noise influences the resulting process model and a feasible process model can be chosen. Third, paths consisting of ordered actions as input can be provided by multiple approaches. Thus, work should be carried out to transfer our approach to related research fields such as web service composition and process model verification which may also benefit from our work. For instance, currently we work on an enhancement of an existing (web) service composition and selection approach by considering feasible parallelizations of services during runtime of a process. Moreover, future work should analyze how our approach can be applied to manually constructed process models to allow the construction of (additional) parallelizations for such models. Our approach forms an appropriate foundation for this as well as for the aforementioned enhancements and thus serves as a suitable basis for further research.

6 References

[1] J.F. Chang, Business process management systems: Strategy and implementation, CRC Press, 2016.

[2] J. vom Brocke, J. Mendling (Eds.), Business Process Management Cases: Digital Innovation and Business Transformation in Practice, Springer International Publishing, Cham, 2018.

[3] T.H. Davenport, Process innovation: reengineering work through information technology, Harvard Business Press, 1993.

[4] IEEE Task Force on Process Mining, Process mining manifesto, in: Business process management workshops, 2012, pp. 169–194.

[5] W.M.P. van der Aalst, Process mining: Data science in action, Springer, Heidelberg, 2016.

[6] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, F.M. Maggi, A. Marrella, M. Mecella, A. Soo, Automated discovery of process models from event logs: Review and benchmark, IEEE Transactions on Knowledge and Data Engineering (2018).

[7] A.L. Lemos, F. Daniel, B. Benatallah, Web service composition: a survey of techniques and tools, ACM Computing Surveys (CSUR) 48 (2016) 33.

[8] Y. Xu, J. Yin, S. Deng, N.N. Xiong, J. Huang, Context-aware QoS prediction for web service recommendation and selection, Expert Systems with Applications 53 (2016) 75–86.

[9] B. Heinrich, A. Schiller, D. Schön, The cooperation of multiple actors within process models: an automated planning approach, Journal of Decision Systems 27 (2018) 238–274. https://doi.org/10.1080/12460125.2019.1600894.

[10] J. Hoffmann, I. Weber, F.M. Kraft, SAP speaks PDDL: Exploiting a software-engineering model for planning in business process management, Journal of Artificial Intelligence Research 44 (2012) 587–632.

[11] M. Henneberger, B. Heinrich, F. Lautenbacher, B. Bauer, Semantic-based planning of process models, in: Multikonferenz Wirtschaftsinformatik, GITO-Verlag, Berlin, 2008, pp. 1677–1689.

[12] B. Heinrich, M. Klier, S. Zimmermann, Automated planning of process models: Design of a novel approach to construct exclusive choices, Decision Support Systems 78 (2015) 1–14.

[13] B. Heinrich, D. Schön, Automated Planning of Context-aware Process Models, in: 23rd European Conference on Information Systems, 2015.

[14] B. Heinrich, D. Schön, Automated Planning of Process Models: The Construction of Simple Merges, in: 24th European Conference on Information Systems, ECIS 2016, Istanbul, Turkey, June 12-15, 2016, 2016, Research Paper 183.

[15] F. Lautenbacher, T. Eisenbarth, B. Bauer, Process model adaptation using semantic technologies, in: IEEE 13th Enterprise Distributed Object Computing Conference Workshops, 2009, Auckland, New Zealand, IEEE, Piscataway, NJ, USA, 2009, pp. 301–309.

[16] M. Ghallab, D. Nau, P. Traverso, Automated planning: theory & practice, Elsevier, 2004.

[17] M. Bortlik, B. Heinrich, M. Mayer, Multi User Context-Aware Service Selection for Mobile Environments, Business & Information Systems Engineering 60 (2018) 415–430.

[18] P. Soffer, Y. Wand, M. Kaner, Conceptualizing routing decisions in business processes: Theoretical analysis and empirical testing, Journal of the Association for Information Systems 16 (2015) 345.

[19] W.M.P. van der Aalst, A.P. Barros, B. Kiepuszewski, A.H.M. Ter Hofstede, Workflow patterns, Distributed and parallel databases 14 (2003) 5–51.

[20] W.M.P. van der Aalst, A.H.M. Ter Hofstede, YAWL: Yet another workflow language, Information systems 30 (2005) 245–275.

[21] Q. He, J. Yan, H. Jin, Y. Yang, Adaptation of web service composition based on workflow patterns, Service-Oriented Computing-ICSOC 2008 (2008) 22–37.

[22] N. Russell, W.M.P. van der Aalst, A.H.M. Ter Hofstede, Workflow Patterns: The Definitive Guide, MIT Press, 2016.

[23] S. Cheikhrouhou, S. Kallel, N. Guermouche, M. Jmaiel, The temporal perspective in business process modeling: A survey and research challenges, Service Oriented Computing and Applications 9 (2015) 75–85.

[24] J.E. Cook, A.L. Wolf, Event-based detection of concurrency, in: ACM SIGSOFT Software Engineering Notes, 1998, pp. 35–45.

[25] M. Alrifai, T. Risse, W. Nejdl, A hybrid approach for efficient Web service composition with end-to-end QoS constraints, ACM Transactions on the Web (TWEB) 6 (2012) 7.

[26] F. Hasić, J. de Smedt, J. Vanthienen, Augmenting processes with decision intelligence: Principles for integrated modelling, Decision Support Systems 107 (2018) 1–12.

[27] T.-F. Kummer, J. Recker, J. Mendling, Enhancing understandability of process models through cultural-dependent color adjustments, Decision Support Systems 87 (2016) 1–12.

[28] W.M.P. van der Aalst, Business process management: A comprehensive survey, ISRN Software Engineering 2013 (2013).

[29] H.A. Reijers, J. Mendling, A study into the factors that influence the understandability of business process models, IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans 41 (2011) 449–462.

[30] N. Russell, A.H.M. Ter Hofstede, W.M.P. van der Aalst, N. Mulyar, Workflow control-flow patterns: A revised view, BPM Center Report BPM-06-22, BPMcenter. org (2006) 6–22.

[31] S. Migliorini, M. Gambini, M. La Rosa, A.H.M. Ter Hofstede, Pattern-based evaluation of scientific workflow management systems, Technical Report, Queensland University of Technology (2011).

[32] J. Wang, A. Kumar, A framework for document-driven workflow systems, in: International Conference on Business Process Management, 2005, pp. 285–301.

[33] T. Haerder, A. Reuter, Principles of transaction-oriented database recovery, ACM Computing Surveys (CSUR) 15 (1983) 287–317.

[34] L. Wen, J. Wang, W.M.P. van der Aalst, B. Huang, J. Sun, A novel approach for process mining based on event types, Journal of Intelligent Information Systems 32 (2009) 163–190.

[35] M.T. Wynn, H.M.W. Verbeek, W.M.P. van der Aalst, A.H.M. Ter Hofstede, D. Edmond, Business process verification-finally a reality!, Business Process Management Journal 15 (2009) 74–92.

[36] B. Wetzstein, Z. Ma, A. Filipowska, M. Kaczmarek, S. Bhiri, S. Losada, J.-M. Lopez-Cob, L. Cicurel, Semantic Business Process Management: A Lifecycle Based Requirements Analysis, in: Workshop on Semantic Business Process and Product Lifecycle Management, CEUR, Germany, 2007, pp. 7–17.

[37] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Planning in nondeterministic domains under partial observability via symbolic model checking, in: Proceedings of the 17th International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 2001, pp. 473–478.

[38] B. Bonet, H. Geffner, GPT: a tool for planning with uncertainty and partial information, in: Proceedings of the ICAI'01 Workshop on Planning with Uncertainty and Partial Information, Seattle, WA, USA, 2001, pp. 82–87.

[39] A.V. Deokar, O.F. El-Gayar, Decision-enabled dynamic process management for networked enterprises, Information Systems Frontiers 13 (2011) 655–668.

[40] B. Heinrich, M. Klier, S. Zimmermann, Automated Planning of Process Models-Towards a Semantic-based Approach, in: S. Smolnik, F. Teuteberg, T. Oliver (Ed.), Semantic Technologies for Business and Information Systems Engineering: Concepts and Applications, 2012, pp. 169–194.

[41] W. Binder, I. Constantinescu, B. Faltings, Service invocation triggers: a lightweight routing infrastructure for decentralised workflow orchestration, International journal of high performance computing and networking 6 (2009) 81–90.

[42] I. Constantinescu, B. Faltings, W. Binder, Large scale, type-compatible service composition, in: IEEE International Conference on Web Services, 2004, 2004, pp. 506–513.

[43] H. Meyer, M. Weske, Automated service composition using heuristic search, in: J. Eder, S. Dustdar (Eds.), Business Process Management, Springer, Wien, 2006, pp. 81–96.

[44] J. Pathak, S. Basu, R. Lutz, V. Honavar, Parallel Web Service Composition in MoSCoE: A Choreography-based Approach, in: IEEE 4th European Conference on Web Services, 2006, IEEE, Los Alamitos, CA, USA, 2006, pp. 3–12.

[45] P. Bertoli, M. Pistore, P. Traverso, Automated composition of web services via planning in asynchronous domains, Artificial Intelligence 174 (2010) 316–361.

[46] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Strong planning under partial observability, Artificial Intelligence 170 (2006) 337–384.

[47] M. Pistore, P. Traverso, P. Bertoli, A. Marconi, Automated synthesis of composite BPEL4WS web services, in: IEEE International Conference on Web Services, 2005, IEEE, Los Alamitos, CA, USA, 2005, pp. 293–301.

[48] F. Lécué, A. Delteil, A. Léger, O. Boissier, Web service composition as a composition of valid and robust semantic links, International Journal of Cooperative Information Systems 18 (2009) 1–62.

[49] A.M. Omer, A. Schill, Web service composition using input/output dependency matrix, in: Proceedings of the 3rd workshop on Agent-oriented software engineering challenges for ubiquitous and pervasive computing, 2009, pp. 21–26.

[50] A.M. Omer, A framework for Automatic Web Service Composition based on service dependency analysis. Dissertation, 2011.

[51] M. Rathore, U. Suman, An Inheritance based Service Execution Planning Approach using Bully Election Algorithm, in: Proceedings of the 2015 International Conference on Advanced Research in Computer Science Engineering & Technology (ICARCSET 2015), 2015, p. 19.

[52] V. Vanitha, V. Palanisamy, K. Baskaran, Automatic Service Graph Generation for Service Composition in Wireless Sensor Networks, Procedia Engineering 30 (2012) 591–597.

[53] T. Madhusudan, N. Uttamsingh, A declarative approach to composing web services in dynamic environments, Decision Support Systems 41 (2006) 325–357.

[54] S.-Y. Hwang, W.-S. Yang, On the discovery of process models from their instances, Decision Support Systems 34 (2002) 41–57.

[55] W.M.P. van der Aalst, T. Weijters, L. Maruster, Workflow mining: Discovering process models from event logs, IEEE Transactions on Knowledge and Data Engineering 16 (2004) 1128–1142.

[56] L. Wen, W.M.P. van der Aalst, J. Wang, J. Sun, Mining process models with non-free-choice constructs, Data Mining and Knowledge Discovery 15 (2007) 145–180.

[57] W.M.P. van der Aalst, Process mining: Overview and opportunities, ACM Transactions on Management Information Systems (TMIS) 3 (2012) 7.

[58] A. Weijters, W.M.P. van der Aalst, A.A. De Medeiros, Process mining with the heuristics miner-algorithm, Technische Universiteit Eindhoven, Tech. Rep. WP 166 (2006) 1–34.

[59] T. Jin, J. Wang, Y. Yang, L. Wen, K. Li, Refactor business process models with maximized parallelism, IEEE Transactions on Services Computing 9 (2016) 456–468.

[60] K. Sycara, M. Paolucci, A. Ankolekar, N. Srinivasan, Automated discovery, interaction and composition of semantic web services, Web Semantics: Science, Services and Agents on the World Wide Web 1 (2003) 27–46.

[61] B. Heinrich, M. Bolsinger, M. Bewernik, Automated planning of process models: the construction of exclusive choices, in: Proceedings of the 30th International Conference on Information Systems, Phoenix, AZ, 2009.

[62] S. Arora, B. Barak, Computational complexity: a modern approach, Cambridge University Press, 2009.

[63] A. Cobham, The intrinsic computational difficulty of functions, in: Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science, North Holland Publishing Co., Amsterdam, 1965, pp. 24–30.

[64] N. Prat, I. Comyn-Wattiau, J. Akoka, A taxonomy of evaluation methods for information systems artifacts, Journal of Management Information Systems 32 (2015) 229–267.

[65] A.R. Hevner, S.T. March, J. Park, S. Ram, Design science in information systems research, MIS quarterly 28 (2004) 75–105.

[66] J.F. Nunamaker, M. Chen, T.D.M. Purdin, Systems development in information systems research, Journal of Management Information Systems 7 (1991) 89–106.

[67] J. Venable, J. Pries-Heje, R. Baskerville, A comprehensive framework for evaluation in design science research, in: International Conference on Design Science Research in Information Systems, 2012, pp. 423–438.

[68] Y. Sun, P.B. Kantor, Cross-Evaluation: A new model for information system evaluation, Journal of the American Society for Information Science and Technology 57 (2006) 614–628.

[69] S.M. Siha, G.H. Saad, Business process improvement: Empirical assessment and extensions, Business Process Management Journal 14 (2008) 778–802.

[70] I. Weber, J. Hoffmann, J. Mendling, Beyond soundness: on the verification of semantic business process models, Distributed and parallel databases 27 (2010) 271–343.

[71] B. Heinrich, M. Mayer, Service selection in mobile environments: considering multiple users and context-awareness, Journal of Decision Systems 27 (2018) 92–122.