Pattern-based action engine: Generating process management actions using temporal patterns of process-centric problems

Gyunam Park [a,*], Daniel Schuster [b], Wil M.P. van der Aalst [a]

[a] Process and Data Science Group (PADS), RWTH Aachen University, Ahornstr. 55, Aachen, 52074, Nordrhein-Westfalen, Germany
[b] Fraunhofer FIT, Schloss Birlinghoven, Konrad-Adenauer-Str., Sankt Augustin, 53757, Nordrhein-Westfalen, Germany

A B S T R A C T

As business environments become more competitive, organizations strive to improve their business processes to reduce costs and increase quality and productivity. As process improvement traditionally embraces manual creative tasks that are time-consuming and labor-intensive, the need for automating it arises. Action-Oriented Process Mining (AOPM) aims to support automated process improvement by leveraging various process mining techniques. To that end, AOPM first monitors the presence of operational constraints, i.e., operational problems, in business processes, e.g., a high waiting time for patients to register. Next, it produces interim management actions designed to address these transient problems by analyzing the monitoring results. For instance, if an excessive waiting time persists for more than a week, the system might recommend dispatching additional resources for the upcoming week. Contrary to the mature process mining support for monitoring operational constraints, the action part is typically missing in today's process mining tools. In this work, we propose an action engine to support the automatic generation of actions. It analyzes temporal patterns of monitoring results and produces action plans that describe the execution of management actions. We have demonstrated a use case using the data of a Dutch financial institute to evaluate the feasibility of the proposed action engine and conducted experiments to evaluate its effectiveness.

1. Introduction

Organizations have endeavored to improve their business processes using various approaches. Business process reengineering aims to systematically manage process improvements by making changes in people, processes, and technology. Hammer (1990) provided a collection of principles for improving business processes derived from case studies of successful business processes. Business process redesign focuses on more neutral changes in terms of size and pace of changes. Reijers and Liman Mansar (2005) propose a framework for business process redesign and 29 best practices based on the framework. Six Sigma/Lean Management aims to focus on identifying process problems and then eliminating them. Whereas Six Sigma identifies process problems by monitoring deviations in a number of process performance measures (Pyzdek and Keller, 2014), Lean Management identifies process problems by analyzing various "wastes" (Martínez-Jurado and Moyano-Fuentes, 2014).

Such efforts to improve business processes commonly require manual activities that are time-consuming, expensive, and labor-intensive (Vanwersch et al., 2015). Beerepoot et al. (2023) enlist automating business process improvements as one of "the biggest business process management problems to solve before we die". The goal is to incorporate process improvement as a part of daily business and incrementally improve processes with increasing degrees of autonomy.

Action-Oriented Process Mining (AOPM) aims to achieve automated and continuous process improvement by leveraging process mining techniques (van der Aalst and Carmona, 2022). Process mining provides a wide range of techniques both for backward-looking (e.g., finding a bottleneck in a process) and forward-looking insights (e.g., predicting a bottleneck). Backward-looking process mining techniques include process discovery, conformance checking, and performance analysis, whereas forward-looking process mining techniques include deviation detection, prediction, and recommendation (van der Aalst, 2016).

The framework for AOPM embodies two major functional components to automatically generate process management actions (Park and van der Aalst, 2022). First, *constraint monitoring* analyzes event data to evaluate the presence of various *operational constraints* in business processes. An operational constraint in business processes represents an operational problem. For instance, an Order-To-Cash (O2C) process

* Corresponding author.
E-mail addresses: gnpark@pads.rwth-aachen.de (G. Park), daniel.schuster@fit.fraunhofer.de (D. Schuster), wvdaalst@pads.rwth-aachen.de (W.M.P. van der Aalst).

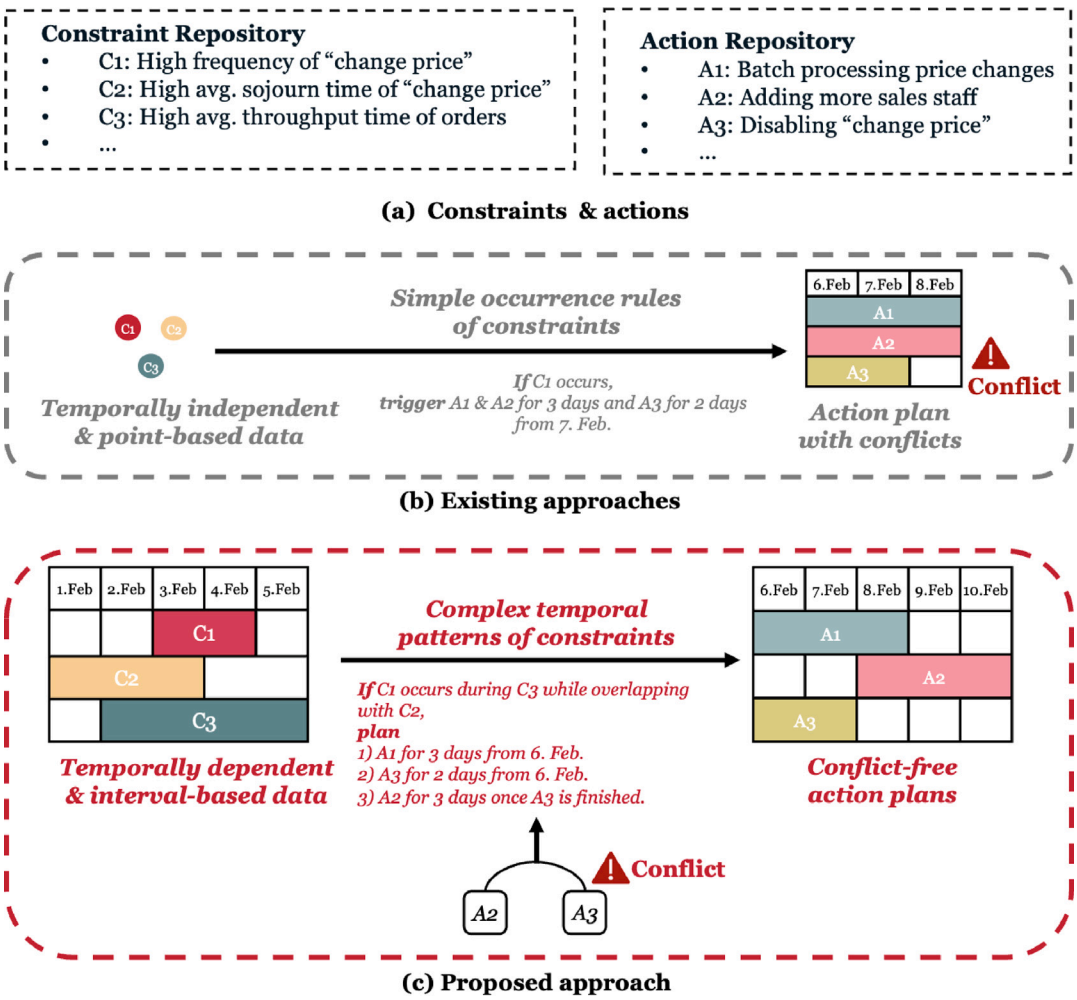


Fig. 1. (a) Examples of operational constraints (i.e., operational problems) and actions in an O2C process, (b) Existing action engines analyzing operational constraints as temporally independent and point-based data and producing conflicting action plans, (c) Our proposed action engine analyzing operational constraints as temporally dependent and interval-based data and producing conflict-free action plans.

may have the operational constraints shown in Fig. 1(a): *C1*, *C2*, and *C3*. The insights from backward-looking process mining techniques can be used to elicit such operational constraints to monitor, and various backward-looking and forward-looking monitoring techniques can be used to track the (potential) presence of such operational constraints. Second, an *action engine* analyzes the monitoring results and generates necessary management actions. For instance, consider actions *A1*, *A2*, and *A3* depicted in Fig. 1(a) to mitigate the risk caused by the presence of operational constraints.

In contrast to extensive support for constraint monitoring, an action engine is missing in most of today's process mining tools. *Celonis Action Engine* is one of the few examples implementing action-oriented process mining (Badakhshan et al., 2019). Based on a trigger-action programming model (Ur et al., 2016), the action engine generates triggering signals by analyzing the event data and executes the actions corresponding to these signals to the source system. For instance, as shown in Fig. 1(b), it analyzes the presence of operational constraint *C2* and, if so, triggers action *A1* and *A2* for 3 days. In Park and van der Aalst (2022), the cube-based action engine is proposed to use multi-dimensional queries to analyze the aggregated value, e.g., if *C1* occurs more than once a week, then trigger actions.

The existing approaches have two limitations. First, they consider operational constraints as temporally independent and point-based data, analyzing relatively simple patterns, e.g., the mere presence of operational constraints. However, in reality, operational constraints are (1) temporally dependent, e.g., *C1* is followed by *C2*, and (2) interval-based, e.g., *C1* is present from 1.Feb. to 2.Feb. By analyzing operational constraints as temporally dependent and interval-based data, we enhance an action engine to analyze the temporal pattern of operational constraints, as shown in Fig. 1(c). Second, the existing approaches consider actions as being conflict-free. However, in reality, actions may have conflicts, i.e., some actions cannot be executed simultaneously. For instance, *A2* and *A3* may have conflicts, i.e., adding more staff to support price changes cannot be simultaneously executed with disabling the activity.

In this work, we propose a *pattern-based action engine* consisting of three phases. First, we design *action graphs* that are graphical notations to specify how to analyze temporal patterns of operational constraints and how to generate the corresponding actions, using, e.g., domain knowledge. Next, based on the action graphs, we generate actions by analyzing *constraint instances*, i.e., the record of constraint monitoring. Finally, we plan the generated actions by considering possible conflicts between the actions. We evaluate the feasibility of the proposed action engine using a real-life loan application process of a Dutch financial institute. Moreover, we conduct experiments to evaluate the scalability and performance of the action engine.

The remainder is organized as follows. We present related work in Section 2. Next, we explain preliminaries in Section 3. Afterward, we present the proposed action engine in Section 4. Section 5 presents the evaluation of the proposed approach. Finally, Section 6 discusses the implication and limitations of the proposed approach, and Section 7 concludes this paper.

2. Related work

This section presents the related work in business process reengineering & redesign, and action-oriented process mining. Furthermore, we introduce several approaches to analyzing temporal patterns of interval-based data. Finally, we present literature on process improvement actions to introduce various available management actions that can incrementally improve business processes.

2.1. Business process reengineering and redesign

The drive to improve business processes is an ongoing pursuit (Dumas et al., 2013). *Business process engineering*, an early systematic approach to this endeavor, aims for transformational changes across people, processes, and technology (Hammer, 1990). It aspires to radically restructure an organization's existing processes, aiming for significant improvements in performance, efficiency, and effectiveness.

In contrast, *business process redesign* places emphasis on more moderate adjustments, with a focus on refining existing processes rather than complete transformation (Mansar and Reijers, 2007). Reijers and Liman Mansar (2005) offer a comprehensive framework for business process redesign, highlighting six key elements to consider during redesign efforts. Based on these elements, the authors share 29 best practices for process redesign, derived from case studies and expert opinions. Fehrer et al. (2022) suggest a tool, Assisted Business Process Redesign (ABPR), to streamline the redesign process. As a data-driven, iterative approach, ABPR targets specific performance objectives. It delivers four tiers of recommendations, with each tier increasing in domain and use case specificity. The proposed reference architecture serves as a template for new instantiations, addressing the current void in available tools.

2.2. Action-oriented process mining

Recent developments suggest various methods for producing prescriptive actions based on predicted performance and risk of process instances. Kubrak et al. (2022) suggest a comprehensive framework to categorize prescriptive process monitoring techniques. One of the core dimensions of this framework is the *objective*, which classifies techniques into two main categories: those focused on reducing negative outcomes, such as defect rates, and those aiming to optimize a specific performance metric, like cycle time. For example, the work of Conforti et al. (2015) fits into the first category with its goal of minimizing risks in ongoing process instances. Conversely, the method by Bozorgi et al. (2021) aligns with the second category, aiming to optimize the cycle time of processes. Another key dimension in the framework is *type of actions* prescribed to meet the objectives. This dimension identifies three primary types of actions: resource allocation, control flow adjustments, and alarm generation. For example, Conforti et al. (2015) deploy techniques that generate actions concerning resource allocation. Their approach predicts the risk levels of running process instances and then optimizes resource allocations based on these predictions. Similarly, Weinzierl et al. (2020) focus on control flow adjustments by recommending the next best activities based on key performance indicators. Lastly, Fahrenkrog-Petersen et al. (2022) belong to the alarm generation category, as their approach aims to produce alarms for predicted problematic instances. Their method further employs a cost model to optimize these alarms, thereby capturing the trade-offs between different types of alarms.

Celonis Action Engine (Badakhshan et al., 2019), a part of *Celonis Execution Management System (EMS)*, is a representative effort to achieve the goal of action-oriented process mining, i.e., turning diagnostics into actions. It analyzes event data to produce signals and execute necessary actions to source systems based on the signals. In Park and van der Aalst (2020), a systematic framework for action-oriented process mining is suggested to transform event data into process-centric diagnostics and the diagnostics into needed management actions instead of generating actions in an ad-hoc manner. Based on the framework, a cube-based action engine (Park and van der Aalst, 2022) is suggested to systematically turn diagnostics to actions. The action engine enables multi-dimensional queries to analyze aggregated diagnostics values and trigger the corresponding action.

Existing approaches in action-oriented process mining, however, consider operational constraints as temporally independent and point-based data, generating actions based on relatively simple occurrence rules (cf. Fig. 1). Moreover, they do not consider possible conflicts between actions that are prevalent in reality when triggering the generated actions. This paper tackles this research gap by proposing a pattern-based action engine to analyze complex temporal patterns of operational constraints and produce conflict-free actions by considering possible conflicts between the actions. Additionally, in contrast to existing prescriptive process monitoring techniques that focus on individual process instances, our proposed action engine operates at the process level, tackling operational constraints across multiple process instances.

2.3. Temporal pattern mining

Operational constraints within business processes often manifest as temporal patterns that indicate problematic situations requiring actions. Domain experts, knowledgeable in their specific processes, are typically responsible for defining these patterns. However, this task may be challenging given the complexity of operational constraints in business processes. One potential solution is the automation of temporal pattern discovery from the historical record of constraint monitoring, thereby facilitating the identification and characterization of these problematic situations. This is where temporal pattern mining, a relatively young research field, plays a role.

Temporal pattern mining aims to analyze temporal patterns in symbolic time intervals, also known as interval-based data. Villafane et al. (2000) propose an initial approach to analyze containments of time intervals in multi-symbolic time interval series. Kam and Fu (2000) use Allen's temporal relations (Allen, 1983) to compose frequent temporal patterns, called *A1 patterns*. An *A1 pattern* represents the temporal relation between an existing temporal pattern and the next symbolic time interval. However, the temporal relations among the time intervals in A1 patterns are ambiguous since the temporal relations are defined only among the pairs of successive intervals (Kam and Fu, 2000). Höppner (2001) presents a nonambiguous representation of temporal patterns by a k^2 matrix representing all of the pair-wise relations within a k-sized Time Intervals Related Pattern (TIRP). Later, the representation was simplified to the sequence of symbolic time intervals and the conjunction of the pair-wise temporal relations among time intervals by removing redundant temporal relations, i.e., the inverse of Allen's temporal relations (Allen, 1983).

Papapetrou et al. (2009) propose two approaches for generating TIRP trees: Breath First Search (BFS)-based and Depth First Search (DFS)-based approaches. The former enumerates the node at each level before proceeding to the next level, whereas the latter enumerates each path up to the leaves as a greedy approach. Moreover, they present a hybrid approach that combines both approaches based on the Sequential Pattern Mining Algorithm (SPAM) (Ayres et al., 2002) mining method. Moskovitch and Shahar (2015) propose the KarmaLego algorithm that efficiently generates candidate TIRPs by exploiting the transitivity property of temporal relations. Harel and Moskovitch (2021) suggest TIRPClo to efficiently discover the complete set of frequent closed TIRPs, i.e., a compact subset of all the frequent TIRPs, from which their complete information can be revealed.

Wu and Chen (2007) propose TPrefixSpan for mining nonambiguous temporal patterns from interval-based events by adopting a sequential mining algorithm PrefixSpan (Pei et al., 2004). The proposed approach represents each time interval as a start-time event and an end-time

event and discovers frequent sequences of the start-time and end-time events.

In Martini et al. (2023), the authors introduce concurrency-aware process execution variants that account for time interval data of activities. As these variants represent a tree structure, frequent patterns can be mined using frequent subtree mining approaches, such as Asai et al. (2004), Chi et al. (2004). The process mining tool Cortado (Schuster et al., 2023) features frequent pattern mining from concurrency-aware process execution variants.

In this work, we use Cortado (Schuster et al., 2023) in the evaluation of our proposed action engine. By exploiting the tailored approach to analyze business processes, we discover frequent temporal patterns of operational constraints and identify relevant problems to tackle with actions generated by our proposed action engine. In particular, the tool supports the discovery of temporal patterns in a binary tree structure that facilitates the interpretation by domain experts. Moreover, the tool is publicly available, ensuring the reproducibility of our evaluation results.

2.4. Process management actions

In this work, we conceptualize an action as a change in Process-Aware Information Systems (PAISs) that include ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), WFM (Workflow Management Systems), and BPMS (Business Process Management Systems) (Dumas et al., 2005). In contrast to data- or function-centered information systems, PAISs are characterized by a strict separation of process logic and application code. The process logic dictates the order and conditions for the execution of various activities within a process, e.g., order confirmations only after order payments. The application code includes business logic, user interface, data manipulation, computations, interactions with other systems, etc. Examples include calculating the total cost of items in an order, processing user authentication, communicating with the payment gateway to process the transaction, and updating the database.

An action corresponds to a change in the process logic or the application code. In the following, we explain actions in process logic and ones in application code.

2.4.1. Actions in process logic

Several approaches have been proposed to change the process logic of PAISs: *process redesign*, *process configuration*, and *process adaptation*. The process logic of PAISs is often described by explicit process models, e.g., Workflow Nets (van der Aalst, 1998) and Business Process Model and Notation (BPMN) (Grosskopf et al., 2009). Such a process model prespecifies activities to be executed, their control flow dependencies, the organizational entities performing the activities, the data objects manipulated by them, and the software applications (e.g., order management systems) needed. Based on a process model, process instances can be created at run-time, each representing a concrete business case (e.g., an order by customer Adams). *Process redesign* aims to improve business processes by modifying process models at design time and, thus, influencing the process instances created at run-time by the model. Control flow patterns are used to modify process models, e.g., by enumerating different alternatives in the process model or by allowing activities to be executed in parallel (Schonenberg et al., 2008).

However, it would be too costly for organizations to design and implement standardized business processes from scratch. For these reasons, there is a great interest in capturing common process knowledge only once and using it in terms of reference process models (reference processes for short). *Process configuration* aims to design a reference process model capturing the behavior of all process variants; i.e., a reference process model merges a multitude of process variants into one configurable model capturing both the commonalities and the differences of the process variants (Rosa et al., 2009). In such a reference process model, variation points are represented in terms of configurable nodes and edges (Rosa et al., 2009). By configuring these, the behavior of the reference process model can be customized to the given context. Modeling languages supporting this approach include Configurable Event-driven Process Chains (C-EPC) (Rosemann and van der Aalst, 2007) and C-YAWL (Gottschalk, 2009).

A business process cannot always be executed in accordance with process models at design time due to emerging exceptions or special situations (Peleg et al., 2009). Hence, authorized users should be allowed to situationally adapt single process instances during run-time to cope with unanticipated exceptions, e.g., by inserting, deleting, or moving activities. *Process adaptations* aim to support such ad hoc deviations from a designed process model while ensuring PAIS robustness to end-users. Reichert and Dadam (1998) present an approach for the support of dynamic structural changes of running process instances. Based upon a formal model called ADEPT, they define a complete and minimal set of change operations called ADEPTflex, which support users in modifying the structure of a running process model while maintaining its structural correctness and consistency. Weber et al. (2008) suggest 18 change patterns independent of concrete implementations and constitute solutions to typical changes. They enable structural changes of process models at a high level of abstraction, e.g., by adding, deleting, or moving activities and process fragments, respectively. Furthermore, adaptation patterns can be applied at both the process type and the process instance level.

2.4.2. Actions in application code

Some approaches focus on revising the application code, specifically the business rules governing the functionality of individual activities, in order to enhance business processes. In the realm of software engineering, such modification initiatives fall under the category of *maintenance* (Buckley et al., 2005). A widely accepted classification of software maintenance identifies 12 types, including training, consultive, evaluative, reformative, updative, groomative, preventive, performance, adaptive, reductive, corrective, and enhancive (Chapin et al., 2001).

Among these, enhancive maintenance activities stand out for their focus on changing business rules to extend or restrict the software's functionality and accessibility. These amendments can involve the addition, elimination, or modification of business rules that dictate the functioning of specific activities within the application. For instance, Wermelinger et al. (2003) propose a framework that permits users to define multiple business rules based on predetermined parameters, thereby enabling the adaptation of these rules in response to changing business needs.

Similarly, Charfi and Mezini (2004) suggest an approach that divides a process into a core part and a business rule part. This division modularizes the business rule segment, enabling it to exist and evolve independently. They achieved this by implementing the business rules as aspects using the BPEL extension AO4BPEL, which can be dynamically (un)deployed during the process interpretation stage.

In summary, various approaches have been proposed in the literature to define actions as changes in PAISs. In this work, we aim to develop a pattern-based action engine that plans such actions by analyzing temporal patterns of operational constraints and resolving conflicts among such actions. The definition of actions is outside the scope of this paper.

3. Background

As depicted in Fig. 1, this work considers operational constraints as temporally dependent and interval-based data, called *constraint instances*. By analyzing *temporal relations* of such constraint instances, we produce *action plans* that describe which actions should be executed and how they should be scheduled to avoid conflicts. In the following, we formally introduce the constraint instances, temporal relations, and action plans.

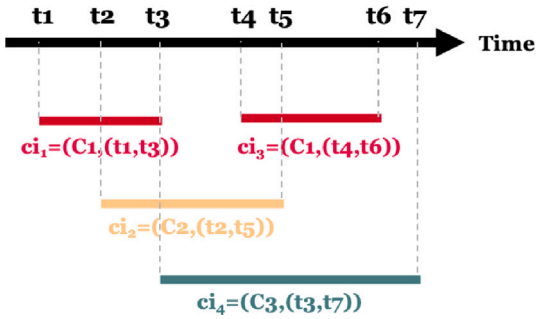


Fig. 2. Examples of constraint instances, $CI_1 = \{ci_1, \ldots, ci_4\} \subseteq \mathbb{U}_{ci}$.

3.1. Constraint instances

An operational constraint in business processes represents an operational problem, e.g., the high frequency of "change price" in an O2C process (cf. *C1* depicted in Fig. 1(a)). Such constraints may occur in a *time window*. A time window describes the period between a start and an end timestamp.

Definition 1 (*Time Window*). Let $\mathbb{U}_{time}$ be the totally ordered set of all possible timestamps. $\mathbb{U}_{tw} = \{(t_s, t_e) \in \mathbb{U}_{time} \times \mathbb{U}_{time} \mid t_s < t_e\}$ is the set of all possible time windows. For any $tw = (t_s, t_e) \in \mathbb{U}_{tw}$, $\pi_{st}(tw) = t_s$ and $\pi_{et}(tw) = t_e$. For any two time windows $tw_1, tw_2 \in \mathbb{U}_{tw}$, $tw_1 \cup tw_2 = (min(\{\pi_{st}(tw_1), \pi_{st}(tw_2)\}), max(\{\pi_{et}(tw_1), \pi_{et}(tw_2)\}))$.

For instance, tw_1 = (2023-01-01 00:00:00, 2023-01-08 00:00:00) is a time window where $\pi_{st}(tw_1)$ = 2023-01-01 00:00:00 and $\pi_{et}(tw_1)$ = 2023-01-08 00:00:00. For tw_1 and tw_2 = (2023-01-08 00:00:00, 2023-01-15 00:00:00), $tw_1 \cup tw_2$ = (2023-01-01 00:00:00, 2023-01-15 00:00:00).

A constraint instance refers to the presence of an operational constraint during a specific time window, e.g., the presence of operational constraint *C1* in Fig. 1(a) from 2023-01-01 to 2023-01-08.

Definition 2 (*Constraint Instance*). Let $\mathbb{U}_{cn}$ be the universe of operational constraint names. $\mathbb{U}_{ci} = \mathbb{U}_{cn} \times \mathbb{U}_{tw}$ is the universe of constraint instances. For $ci = (cn, tw) \in \mathbb{U}_{ci}$, $\pi_{cn}(ci) = cn$ and $\pi_{tw}(ci) = tw$.

For instance, $(C1, tw_1) \in \mathbb{U}_{ci}$ denotes a constraint instance representing the presence of operational constraint *C1* during time window tw_1.

Fig. 2 shows a set of constraint instances, $CI_1 = \{ci_1, \ldots, ci_4\} \subseteq \mathbb{U}_{ci}$. For instance, $ci_1 = (C1, (t1, t3))$ denotes a constraint instance representing the presence of operational constraint *C1* from $t1$ to $t3$.

3.2. Temporal relations

A temporal relation indicates a relation between two time windows, e.g., a time window *overlaps with* the other time window.

Definition 3 (*Temporal Relations*). $Rel \subseteq \mathbb{U}_{tw} \times \mathbb{U}_{tw}$ is a temporal relation. We denote $\mathcal{R}$ to be the set of all possible temporal relations. Any two time windows belong to exactly one relation, i.e., $\forall_{(tw_1,tw_2)\in\mathbb{U}_{tw}\times\mathbb{U}_{tw}} \exists_{Rel\in\mathcal{R}} \forall_{Rel'\in\mathcal{R}\setminus\{Rel\}} (tw_1, tw_2) \in Rel \wedge (tw_1, tw_2) \notin Rel'$.

Allen (1983) suggests 13 temporal relations as shown in Fig. 3. For instance, $before \in \mathcal{R}$ is a temporal relation describing that a time window is before the other time window, i.e., $before = \{(tw_1, tw_2) \in \mathbb{U}_{tw} \times \mathbb{U}_{tw} \mid \pi_{et}(tw_1) < \pi_{st}(tw_2)\}$. Moreover, $overlaps \in \mathcal{R}$ is a temporal relation describing that a time window overlaps with the other time window, i.e., $overlaps = \{(tw_1, tw_2) \in \mathbb{U}_{tw} \times \mathbb{U}_{tw} \mid \pi_{st}(tw_1) < \pi_{st}(tw_2) < \pi_{et}(tw_1) < \pi_{et}(tw_2)\}$.

Any two constraint instances have exactly one temporal relation. For example, $ci_1 = (C1, (t1, t3))$ is before $ci_3 = (C1, (t4, t6))$ as $((t1, t3), (t4, t6)) \in before$. Moreover, $ci_1 = (C1, (t1, t3))$ overlaps with $ci_2 = (C2, (t2, t5))$ as $((t1, t3), (t2, t5)) \in overlaps$. We analyze such temporal relations to produce *action plans*.

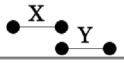
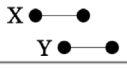
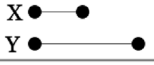
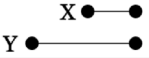
Relation	Relation for Inverse	Example
X before Y	Y after X	
X equal Y	Y equal X	
X meets Y	Y met by x	
X overlaps Y	Y overlapped by X	
X during Y	Y contains X	
X starts Y	Y started by X	
X finishes Y	Y finished by X	

Fig. 3. Allen's temporal relations between two time windows.

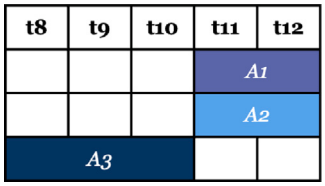


Fig. 4. An example of action plans, AP_1.

3.3. Action plans

An *action plan* describes the execution of actions (e.g., *A1* is executed from 6. Feb.2023 to 8. Feb.2023 and *A3* is executed from 8. Feb.2023 to 11. Feb.2023). We deliberately abstract from specific definitions of actions, denoting $\mathbb{U}_{action}$ as the collection of actions introduced in Section 2.4.

Definition 4 (*Action Plan*). Let $\mathbb{U}_{action}$ be the universe of actions. $AP \subseteq \mathbb{U}_{action} \times \mathbb{U}_{tw}$ is an action plan. We denote $\mathbb{U}_{AP}$ to be the set of all possible action plans.

For example, Fig. 4 shows an action plan, $AP_1 = \{(A1, (t11, t12)), (A2, (t11, t12)), (A3, (t8, t10))\}$, specifying that *A3* should be executed from $t8$ to $t10$, while *A1* and *A2* should be performed from $t10$ to $t12$.

In the following, we explain how a *pattern-based action engine* produces such action plans by analyzing temporal relations inherent in constraint instances.

4. Pattern-based action engine

This section introduces a pattern-based action engine. Fig. 5 provides an overview of the proposed approach, which is composed of three phases. First, domain experts design *action graphs*. These are graphical notations that specify how to analyze temporal patterns of operational constraints and subsequently generate appropriate actions. Each action graph is constituted by *temporal pattern trees* and *actions*. A temporal pattern tree represents the specific temporal pattern of operational constraints being addressed, while an action represents

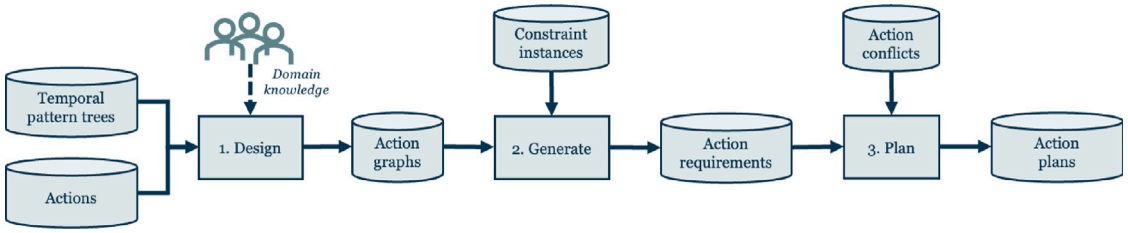


Fig. 5. Overview of the proposed pattern-based action engine.

the corrective measure undertaken in response to that pattern. Subsequently, the pattern-based action engine generates *action requirements* by analyzing constraint instances using the action graphs. An action requirement describes requisite actions, along with their duration. Lastly, the pattern-based action engine formulates action plans by scheduling the requisite actions while taking into account the duration and *action conflicts*.

4.1. Designing action graphs

In this work, we use temporal pattern trees to represent temporal relations among a variable number of operational constraints. Leaves represent operational constraints, and inner nodes represent temporal relations between their subtrees.

Definition 5 (*Temporal Pattern Tree Syntax*)**.** A temporal pattern tree $pt = (V, E, \lambda, r)$ consists of a totally ordered set of nodes V, a set of edges $E \subseteq V \times V$, a labeling function $\lambda \in V \to \mathbb{U}_{cn} \cup \mathcal{R}$, and a root node $r \in V$.

- $(\{v\}, \emptyset, \lambda, v)$ with $\lambda(v) \in \mathbb{U}_{cn}$ is a temporal pattern tree.
- Given two different temporal trees $pt_1 = (V_1, E_1, \lambda_3, r_1)$ and $pt_2 = (V_2, E_2, \lambda_2, r_2)$ with $r \notin V_1 \cup V_2$, $pt = (V, E, \lambda, r)$ is a temporal pattern tree such that
  - $V = V_1 \cup V_2 \cup \{r\}$
  - $E = E_1 \cup E_2 \cup \{(r, r_1), (r, r_2)\}$
  - $\lambda(v) = \lambda_i(v)$ for any $i \in \{1, 2\}$ and $v \in V_i$
  - $\lambda(r) \in \mathcal{R}$

$\mathbb{U}_{pt}$ denotes the set of all possible temporal pattern trees.

Fig. 6 shows a temporal pattern tree pt_0. The tree describes that *C1* "a high frequency of *change price*" happens during *C3* "a high avg. throughput time of orders". This pattern of *C1* and *C3* overlaps with *C2* "a high avg. sojourn time of *change price*".

Note that every temporal relation (inner node) has exactly two children. Next to the graphical representation, any temporal pattern tree can be textually represented because of its totally ordered node set, e.g., $pt_0 \mathrel{\widehat{=}} overlaps(during(C1, C3), C2)$.

For a tree $pt = (V, E, \lambda, r)$, $L(pt) = \{v \in V \mid \nexists_{v' \in V} (v, v') \in E\}$ denotes its *leaf nodes*. Assuming that for each node, the left child appears first in the order, we define the left child of a node $v \in V$ as:

$$lc(pt, v) = \begin{cases} w & \text{if } (v, w) \in E \wedge \nexists_{u \in V, u < w} (v, u) \in E \\ \emptyset & \text{otherwise.} \end{cases}$$

Then, assuming that E^+ denotes the transitive closure E, the leaf nodes of the left child of a vertex $v \in V$ are defined as follows:

$$LL(pt, v) = \begin{cases} \{w \in V \mid (lc(pt, v), w) \in E^+ \wedge \nexists_{u \in V} (w, u) \in E\} & \text{if } lc(v) \neq \emptyset \\ \emptyset & \text{otherwise.} \end{cases}$$

For instance, $lc(pt_0, n_0) = n_{1.1}$ and $LL(pt_0, n_0) = \{n_{2.1}, n_{2.2}\}$. We define $RL(pt, v)$ to be the leaf nodes of the right child of $v \in V$ in the same manner.

We define the semantics of temporal pattern trees in relation to a set of constraint instances with the notion of *occurrences*.

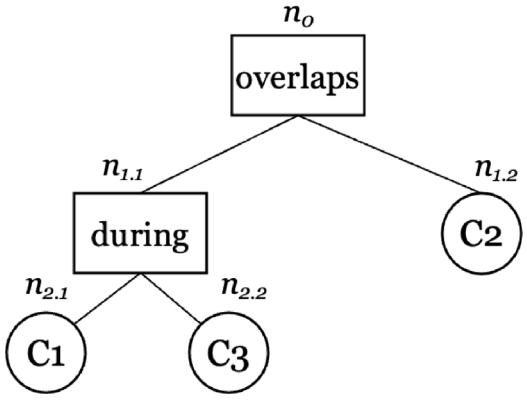


Fig. 6. An example temporal pattern tree $pt_0 = (\{n_0, n_{1.1}, n_{1.2}, n_{2.1}, n_{2.2}\}, \{(n_0, n_{1.1}), \ldots, (n_{1.1}, n_{2.2})\}, \lambda_0, n_0)$ with $\lambda_0(n_0) = overlaps, \ldots, \lambda_0(n_{2.2}) = C3$, where *C1* denotes a high frequency of "change price", *C2* denotes a high avg. sojourn time of "price change", and *C3* denotes a high avg. throughput time of orders (cf. Fig. 1(a)).

Definition 6 (*Temporal Pattern Tree Semantics*)**.** Let $pt = (V, E, \lambda, r) \in \mathbb{U}_{pt}$ be a temporal pattern tree. Let $CI \subseteq \mathbb{U}_{ci}$ be a set of constraint instances. The function $occurs(pt, CI) = true$ if there is an injective mapping $\mu \in L(pt) \to CI$ such that

- $\forall_{v \in L(pt)}\ \lambda(v) = \pi_{cn}(\mu(v))$ and
- $\forall_{v \in V \setminus L(pt)} \left(\bigcup_{v' \in LL(pt,v)} \pi_{tw}(\mu(v')), \bigcup_{v' \in RL(pt,v)} \pi_{tw}(\mu(v')) \right) \in \lambda(v)$.

Otherwise, $occurs(pt, CI) = false$.

Fig. 7 shows an example of an injective mapping μ_1 that demonstrates that $occurs(pt_0, CI_1) = true$. Formally, $\mu_1 = \{(n_{2,1}, ci_3), (n_{2.2}, ci_4), (n_{1.2}, ci_2)\}$ such that

- $\lambda(n_{2.1}) = \pi_{cn}(ci_3) = C1$, $\lambda(n_{2.2}) = \pi_{cn}(ci_4) = C3$, $\lambda(n_{1.2}) = \pi_{cn}(ci_2) = C2$,
- $(\pi_{tw}(ci_3), \pi_{tw}(ci_4)) \in during$, and
- $(\pi_{tw}(ci_3) \cup \pi_{tw}(ci_4), \pi_{tw}(ci_2)) \in overlaps$.

An action graph is a graphical notation to formally represent (1) how to analyze constraint instances using temporal pattern trees and (2) how to trigger actions. It consists of two types of nodes, i.e., temporal pattern trees and actions, and edges connecting the nodes (cf. Fig. 8). Each edge represents that the associated pattern is addressed by the corresponding action and is labeled with the required time duration for the remedial action.

Definition 7 (*Action Graph*)**.** An action graph is a graph $ag = (PT, A, PA, l)$ where

- $PT \subseteq \mathbb{U}_{pt}$ is a set of temporal pattern trees,
- $A \subseteq \mathbb{U}_{action}$ is a set of actions,
- $PA \subseteq PT \times A$ is a set of pattern-to-action edges, and
- $l \in PA \to \mathbb{R}$ maps pattern-to-action edges to time duration.

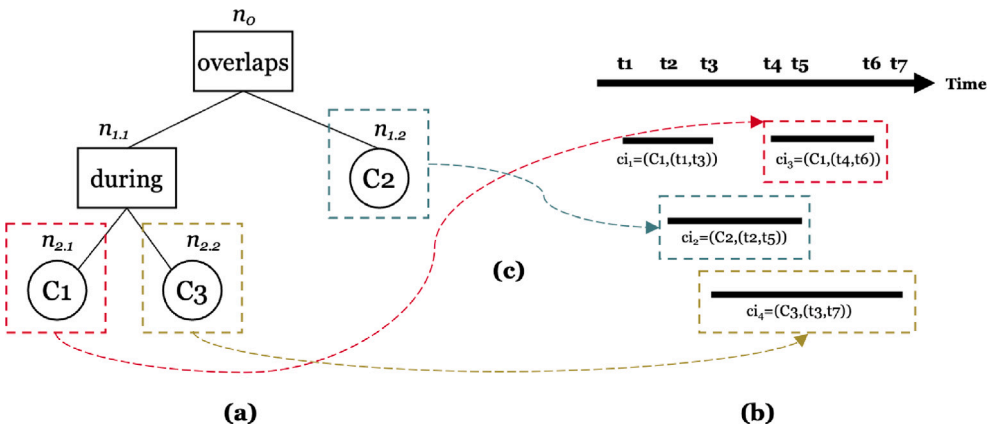


Fig. 7. (a) Temporal pattern tree pt_0 (cf. Fig. 6), (b) Set of constraint instances CI_1 (cf. Fig. 2), and (c) An injective mapping demonstrating that the temporal pattern tree *occurs* in the set of constraint instances.

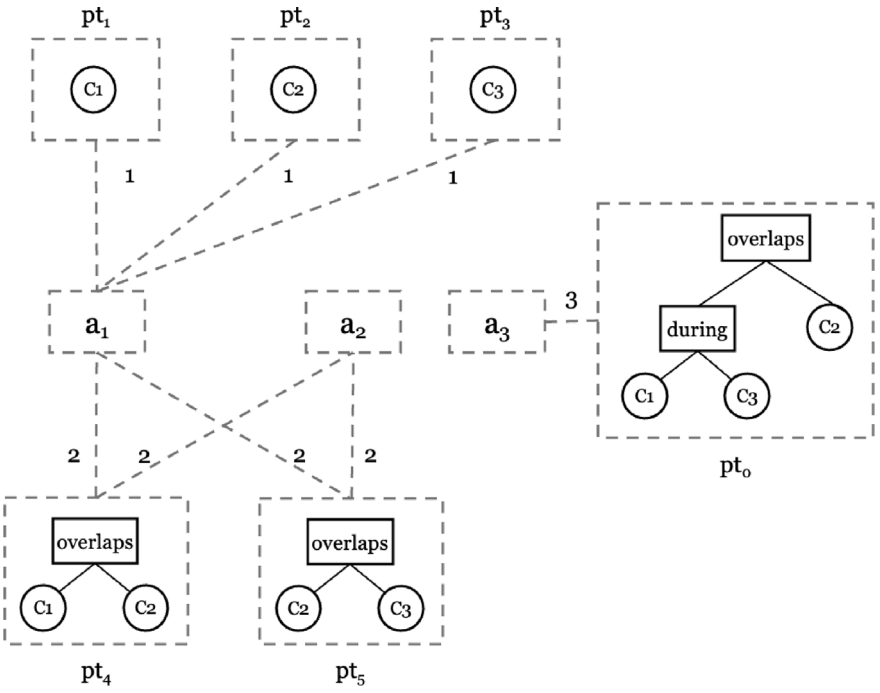


Fig. 8. Action graph ag_1 consisting of six temporal patterns of operational constraints, i.e., $pt_0, \ldots, pt_5 \in \mathbb{U}_{pt}$, and three actions, i.e., $a_1, a_2, a_3 \in \mathbb{U}_{action}$.

$\mathbb{U}_{ag}$ denotes the set of all possible action graphs.

Fig. 8 shows an action graph ag_1. It consists of six problems patterns, i.e., $pt_0, \ldots, pt_5 \in \mathbb{U}_{pt}$, and three actions, i.e., $a_1, a_2, a_3 \in \mathbb{U}_{action}$. Each action reflects a different conceptual action type that we introduce in Section 2.4.

Action a_1, i.e., batch processing price changes, for one day, addresses pt_1, pt_2, pt_3. This action aligns with the concept of *process redesign*, as it involves modifying the process to accumulate price changes and apply them in batches rather than making individual changes. Next, patterns pt_4 and pt_5 are tackled by a_2, i.e., adding more sales staff to support the price changes for two time units, as well as a_1 for two time units. The action a_2 aligns with the concept of *enhancive maintenance*, which involves changing the application code to extend functionality, such as managing and assigning tasks to an increased number of sales staff. Finally, pt_0 is addressed with a_3, i.e., disabling "change" price, for three time units. This action aligns with the concept of *process configuration*, as it involves disabling or restricting the accessibility of an activity.

Formally, $ag_1 = (PT_1, A_1, PA_1, l_1)$ where $PT_1 = \{pt_0, \ldots, pt_5\}$, $A_1 = \{a_1, a_2, a_3\}$, $PA_1 = \{(pt_1, a_1), \ldots, (pt_0, a_3)\}$, $l_1(pt_1, a_1) = 1$, $l_1(pt_0, a_3) = 3$, etc.

4.2. Generating actions

Based on action graphs, we analyze constraint instances to generate *action requirements*. An action requirement describes actions with

required durations. For example, a_1 is required for two time units, a_2 is required for two time units, and a_3 is required for three time units. An action requirement refers to unique actions, i.e., it does not describe the same action with different duration (e.g., a_1 required for two time units and a_1 required for three time units).

Definition 8 (*Action Requirement*)**.** An action requirement $AR \subseteq \mathbb{U}_{action} \times \mathbb{R}$ is a set of action and duration pairs such that, for any $(a, r), (a', r') \in AR$, $a = a' \implies (a, r) = (a', r')$. $\mathbb{U}_{AR}$ denotes the set of all possible action requirements.

For instance, $AR_1 = \{(a_1, 2), (a_2, 2), (a_3, 3)\} \in \mathbb{U}_{AR}$ refers to an action requirement that describes the execution of action a_1 for two time units, the execution of action a_2 for two time units, and the execution of action a_3 for two time units. Note that, for the sake of brevity, we define durations as real numbers.

Given a set of constraint instances and an action graph, an action generator returns an action requirement. More in detail, it generates actions if any of their connected temporal patterns *occur* in the set of constraint instances, e.g., a_1 is generated if any of $pt_1, \ldots, pt_5$ *occur* in a set of constraint instances. In this work, we associate the duration of a generated action with the maximum duration among the ones specified by occurring temporal patterns, e.g., a_1's duration is 2 if both pt_1 and pt_5 *occur*.

Definition 9 (*Action Generator*)**.** An action generator $gen \in \mathcal{P}(\mathbb{U}_{ci}) \times \mathbb{U}_{ag} \to \mathbb{U}_{AR}$ maps a set of constraint instances and an action graph to an action requirement. For any $CI \subseteq \mathbb{U}_{ci}$ and $ag = (PT, A, PA, l) \in \mathbb{U}_{ag}$, $gen(CI, ag) = \{(a, dur) \in \mathbb{U}_{action} \times \mathbb{R} \mid (\exists_{(pt,a)\in PA}\ occurs(pt, CI) = true \wedge dur = l(pt, a)) \wedge (\nexists_{(pt',a)\in PA}\ occurs(pt', CI) = true \wedge dur < l(pt', a))\}$.

Given $CI_1 \subseteq \mathbb{U}_{ci}$ described in Fig. 7 and ag_1 depicted in Fig. 8, $gen(CI_1, ag_1) = AR_1 = \{(a_1, 2), (a_2, 2), (a_3, 3)\}$. First, $(a_1, 2)$ is generated since pt_1, pt_2, pt_3, pt_4, and pt_5 *occur* in CI_1, and the maximum duration specified in the action graph is 2. Second, $(a_2, 2)$ is generated since pt_4, and pt_5 *occur* in CI_1, and the maximum duration specified in the action graph is 2. Finally, $(a_3, 3)$ is generated since $occurs(pt_0, CI_1) = true$, and the maximum duration specified in the action graph is 3.

4.3. Planning actions

Based on action requirements, we plan the execution of actions in the action requirement, e.g., a_1 is executed from 6. Feb.2023 to 8. Feb.2023, and a_3 is executed from 8. Feb.2023 to 11. Feb.2023, ensuring the resolution of any *conflicts* between actions. A conflict between actions can be interpreted in multiple ways, and for this work, we categorize them as follows:

- Temporal Conflicts: These conflicts stem from the timing and scheduling of actions. They can be further classified into:
  - Concurrent Execution Conflicts: These occur when two or more actions are planned to happen simultaneously. If action *a1* involves shutting down a machine for maintenance and action *a2* involves running a diagnostic test on the same machine, scheduling both actions at the same time would create a conflict.
  - Sequential Execution Conflicts: These occur when the order of action execution is essential, such as when one action must conclude before another begins. If action *a3* requires the results of action *a1* for its execution, then *a1* must complete before *a3* starts.
- Effect-based Conflicts: These conflicts emerge based on the aftermaths of the actions. They can be further segmented into:
  - Incompatibility Conflicts: These conflicts arise when the execution of one action nullifies the effect of another. If action *a4* is meant to increase system security but action *a5* inadvertently creates a security vulnerability, the two actions are in conflict.
  - Counterproductivity Conflicts: These conflicts appear when the outcome of an action interferes with or obstructs the overall objective. If the objective is to reduce energy consumption and action *a6* reduces energy use but action *a7* increases it, this creates a counterproductivity conflict.
- Dependency-based Conflicts: These conflicts emerge due to the dependency relationships between actions. They can be further broken down into:
  - Interdependency Conflicts: These conflicts arise when certain actions must be carried out together for a successful operation. If action *a8* sets a system configuration that action *a9* relies on, failing to execute both actions together would result in a conflict.

In this work, our primary focus lies in resolving temporal conflicts among actions. To this end, an action conflict model is designed to express the temporal conflict between actions and precedence constraints. For instance, an action conflict may describe that a_3 has a conflict with a_1, i.e., disabling *change price* cannot be executed simultaneously with alerting sales staff responsible for *change price*, and the conflict between a_3 and a_1 is resolved by executing a_3 before a_1.

Definition 10 (*Action Conflict*)**.** An action conflict $AP = (A, C)$ is a directed acyclic graph where $A \subseteq \mathbb{U}_{action}$ and $C \subseteq A \times A$ such that $(a, a) \notin C^+$ for all $a \in A$, where C^+ denotes the transitive closure of C. $\mathbb{U}_{ac}$ denotes the set of all possible action conflicts.

Fig. 9(b) shows an example action conflict, $ac_1 = (A_1, C_1)$ with $A_1 = \{a_1, a_2, a_3\}$ and $C_1 = \{(a_3, a_1), (a_3, a_1)\}$. Action a_3 has a conflict with action a_1, and the conflict is resolved by executing a_3 before a_1. Moreover, action a_3 has a conflict with action a_2, and the conflict is resolved by executing a_3 before a_2.

Given an action requirement and an action conflict, an action planner produces an action plan. For each action specified in the action requirement, the action planner produces an action instance that describes the execution of the corresponding action with a start timestamp and an end timestamp. While doing so, the action planner ensures that no conflicts exist between the action instances using the action conflict, i.e., no conflicting actions are simultaneously executed.

Definition 11 (*Action Planner*)**.** An action planner $plan \in \mathbb{U}_{AR} \times \mathbb{U}_{ac} \to \mathbb{U}_{AP}$ maps an action requirement and an action conflict to an action plan. For any $AR \in \mathbb{U}_{AR}$ and $(A, C) \in \mathbb{U}_{ac}$,

- $\{(a, \pi_{et}(tw) - \pi_{st}(tw)) \mid (a, tw) \in plan(AR, ac)\} = AR$ and
- $\forall_{(a,a')\in C} \forall_{(a,tw),(a',tw')\in plan(AR,ac)}\ \pi_{et}(tw) \leq \pi_{st}(tw')$.

Fig. 9 shows three action planners that produce different action plans, given action requirement AR_1 and action conflict ac_1. For three actions specified in AR_1, action planner $plan_1$ produces an action plan with three action instances, i.e., $plan_1(AR_1, ac_1) = \{(a_1, (4, 5)), (a_2, (6, 7)), (a_3, (1, 3))\}$. Action instance $(a_1, (3, 5))$ is produced for $(a_1, 2) \in AR_1$, while $(a_2, (5, 7))$ and $(a_3, (0, 3))$ are produced for $(a_2, 2)$ and $(a_3, 3)$, respectively. Actions a_1 and a_2 are planned after a_3 to resolve the conflict specified in ac_1.

In this work, we implement an action planner that minimizes the makespan, i.e., the total time taken to finish all action instances (e.g., the leftmost action plan in Fig. 9(c) has the makespan of 7). By minimizing the makespan, we ensure that actions are executed in the shortest possible time, which in turn enables a faster response to operational issues. However, makespan minimization is not the only possible objective for an action planner. Depending on the specific requirements and constraints of the process and problem scenario,

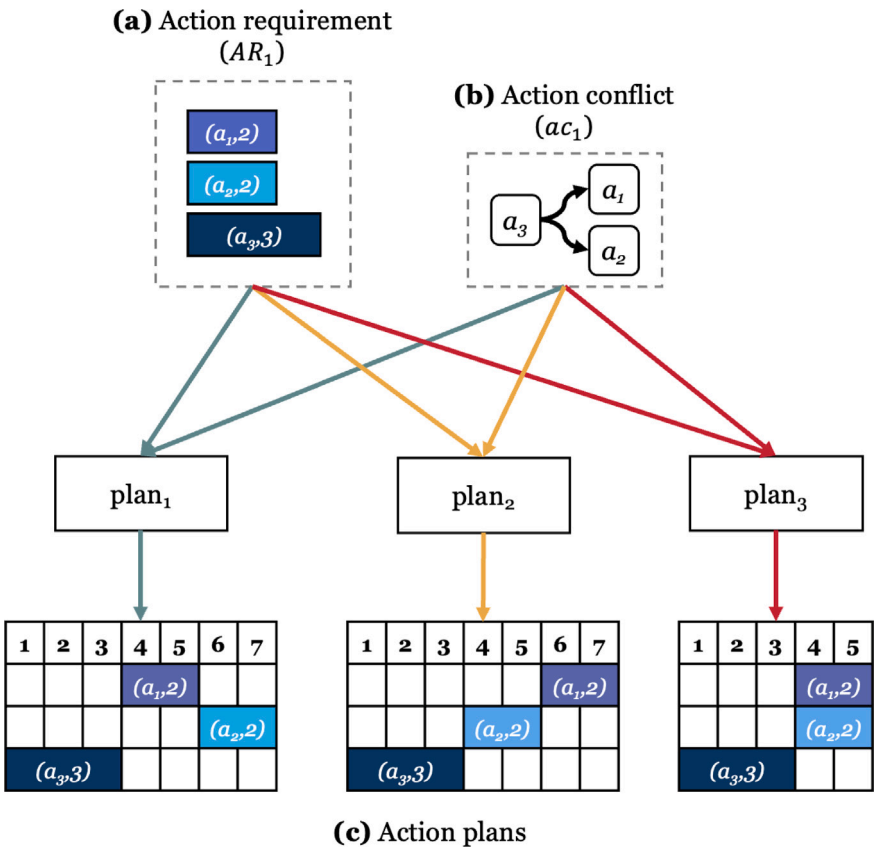


Fig. 9. Examples of action planners: (a) An action requirement (AR_1), (b) An action conflict (ac_1), and (c) Different action plans computed from the action requirements and action conflict.

other potential objectives could include minimizing the total cost of actions, maximizing the quality or effectiveness of actions, etc. For instance, in scenarios where action costs are a concern, an action planner might seek to minimize the total cost of actions. On the other hand, in cases where the quality of the actions is paramount, the planner might aim to maximize the overall effectiveness of the actions, potentially at the expense of longer makespan or higher costs.

We formulate the problem as a parallel machine scheduling problem, i.e., scheduling n jobs (i.e., n actions in the given action requirement) on m parallel machines (i.e., m entities which execute actions) under precedent constraints (i.e., the action conflict), while minimizing the makespan (denoted as $P_m|prec|C_{max}$) (Pinedo, 2012).

$P_m|prec|C_{max}$, is an NP-hard problem (Cheng and Sin, 1990). However, a polynomial time algorithm to solve the problem exists if there are an unlimited number of machines (i.e., $m \geq n$) (Pinedo, 2012). We implement an action planner under the assumption that there are an unlimited number of entities that can execute the actions. Algorithm 1 describes the action planner algorithm. Given an action requirement and an action conflict, we schedule the actions in the action requirement that have no conflicts one at a time, starting at time 0. Whenever an action has been completed, we schedule the actions of which all predecessors have been completed.

The proposed algorithm produces the third action plan described in Fig. 9. The action plan has the minimum makespan of 5 time units, given the action requirement and action conflict.

Algorithm 1 Action planner algorithm

Input: An action requirement $AR \in \mathbb{U}_{AR}$ and an action conflict $(A, C) \in \mathbb{U}_{ac}$
Output: An action plan $AP \in \mathbb{U}_{AP}$

$AP \leftarrow \emptyset$
$S \leftarrow \emptyset$ ▷ A set of pairs of a scheduled action and its completion time
$t \leftarrow 0$
while $AR \neq \emptyset$ **do**
  for $(a, r) \in AR$ **do**
    if $\{a' \mid (a', a) \in C\} \subseteq complete(S, t)$ **then** ▷ *complete* is a function to return the set of completed actions at t
      $S \leftarrow S \cup (a, t + r)$
      $AP \leftarrow AP \cup (a, (t, t + r))$
      $AR \leftarrow AR \setminus \{(a, r)\}$
    end if
  end for
  $t \leftarrow t + 1$
end while
return AP

5. Evaluation

This section evaluates the action engine proposed in Section 4. First, we evaluate the feasibility of the proposed action engine by demonstrating a use case using the data of a real-life loan application process. Next, we conduct experiments to evaluate the scalability and performance of the proposed action engine.

5.1. Use case: Loan application process

We demonstrate a use case using the data of a real-life loan application process from a Dutch financial institute (van Dongen, 2017). The execution of the loan application process involves an application and a

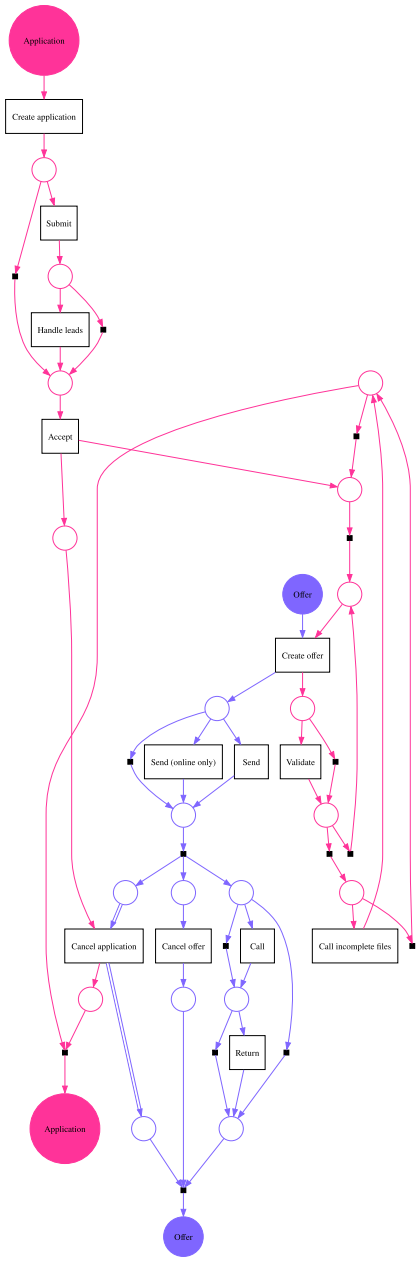


Fig. 10. A process model of the loan application process using object-centric Petri net as a formalism.

variable number of offers. Fig. 10 shows a process model as an object-centric Petri net (van der Aalst and Berti, 2020). First, a customer creates an application by visiting the bank or using an online system. In the former case, *submit* activity is skipped. After the completion and acceptance of the application, the bank offers loans to the customer by sending the offer to the customer and making a call. An offer is either accepted or canceled.

In this use case, we focus on the offers canceled due to various reasons. In order to introduce operational constraints in the context of canceled applications, we explain a process execution with a canceled application as depicted in Fig. 11. The process execution starts with *Application1*. The application has its own lifecycle that consists of multiple activities such as *create application*, *submit*, *call incomplete files*, etc. The process execution involves two offers: *Offer1* and *Offer2*. The lifecycle of each offer includes activities such as *create offer*, *send offer*, *call applicants*, etc. The process execution finishes by canceling *Application1* with *Offer1* and *Offer2*.

During the process, we monitor the presence of the following operational constraints focusing on *cancel application (CA)* activity in the process:

1. *High ratio of CA's executions (**CE**)* which represents the high ratio of applications that are canceled.
2. *High ratio of CA's executions with multiple offers (**MO**)* which represents the high ratio of *CA* events that involve multiple offers.
3. *High avg. flow time of CA (**FT**)* which represents the high avg. *flow time* of *CA* events; In Fig. 11, *flow time* of *CA* is the time difference between the completion of *CA* and the completion of *Application1*'s lifecycle before *CA*.
4. *High avg. sojourn time of CA (**SoT**)* represents the high avg. sojourn time of *CA* events; In Fig. 11, *sojourn time* of *CA* is the time difference between the completion of *CA* and the completion of *Offer2*'s lifecycle before *CA*.
5. *High avg. synchronization time of CA (**SyT**)* represents the high avg. synchronization time of *CA* events; In Fig. 11, *synchronization time* of *CA* is the time difference between the completion of *Offer2*'s lifecycle before *CA* and the completion of *Application1*'s lifecycle before *CA*.
6. *High avg. pooling time of CA w.r.t. offers (**PT**)* represents the high avg. pooling time of *cancel application* events w.r.t. offers; In Fig. 11, *pooling time* of *CA* w.r.t. *offers* is the time difference between the completion of *Offer2*'s lifecycle before *CA* and the completion of *Offer1*'s lifecycle before *CA*.
7. *High avg. readiness time of CA w.r.t. applications (**RT**)* represents the high avg. readiness time of *cancel application* events w.r.t. applications; In Fig. 11, *readiness time* of *CA* w.r.t. *appliications* is the time difference between the completion of *Offer2*'s lifecycle before *CA* and the completion of *Application1*'s lifecycle before *CA*.

5.1.1. Designing action graphs

First, we define relevant temporal patterns of the operational constraints in the loan application process. Typically, these temporal patterns are derived from the expertise of domain experts. The application of frequent temporal pattern mining approaches aids these experts in specifying such patterns, as well as facilitates the validation of their occurrences in the historical process executions.

To discover these temporal patterns, we utilize the weekly monitoring results, spanning from January 2016 to September 2016. For this, we employ the frequent pattern mining approach implemented in a process mining tool (Schuster et al., 2023). Each weekly monitoring result comprises a week's constraint instances, for example, the week starting from 01.01.2016 00:00:00 and ending at 01.07.2016 23:59:59.

From the 91 discovered frequent temporal patterns, we select those we aim to address. Our selection criteria are based on: (a) relevance to the business process, and (b) simplicity and interpretability of the patterns. In terms of (a), we exclude patterns such as $before(FT, CE)$, which refers to a high average flow time of CA *before* a high ratio of CA's executions, as they do not significantly contribute to understanding the problem areas of the loan application process.

In terms of (b), we disregard overly complex patterns such as $before(overlaps(CE, overlaps(MO, FT)), before(overlaps(SoT, SyT), overlaps(SoT, SyT)))$, despite their potential interest. These intricate conditions are too complex to elicit meaningful actions for tackling the problem.

However, it is important to note that these temporal patterns should ideally be confirmed by domain experts for maximum utility. Given our reliance on publicly available data and the unavailability of access to such experts in this evaluation, we have selected the most plausible temporal patterns based on the aforementioned criteria.

Fig. 12 shows the selected six frequent temporal patterns: $pt_1^{loan}, \ldots, pt_6^{loan}$.

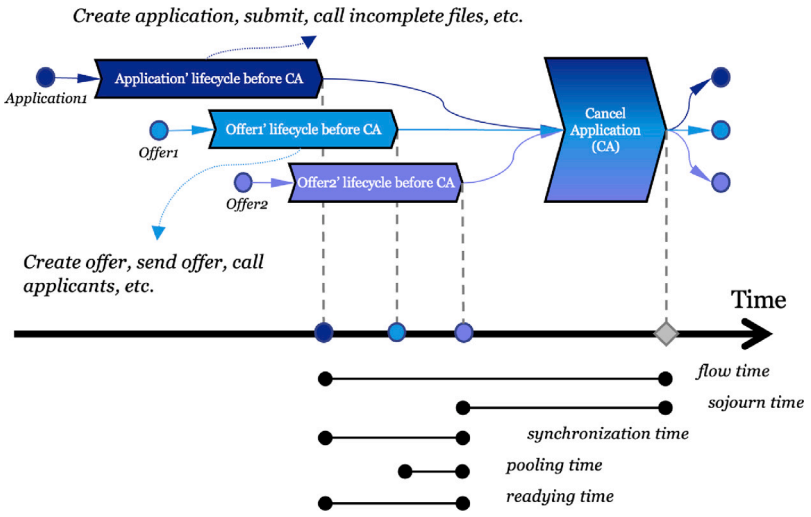


Fig. 11. A process execution of the loan application process.

- pt_1^{loan} portrays a situation where numerous applications are canceled (*CE*) amid a high average flow time for cancellation (*FT*). This pattern indicates the demands for cancellation are increasing, while the cancellation process is delayed, which can lead to customer dissatisfaction.
- pt_2^{loan} illustrates a scenario where multiple offers are presented (*MO*) in many canceled applications (*CE*), concurrent with a high average flow time for the cancellation (*FT*). This could mean unnecessary work is being done for applications destined for cancellation.
- pt_3^{loan} signifies a situation where a high synchronization time of CA (*SyT*) overlaps with a high sojourn time (*SoT*). This could indicate a bottleneck in the offering system, which eventually causes a bottleneck in the application management system.
- pt_4^{loan} represents a repeating occurrence of a high synchronization time (*SoT*) overlapping with a high sojourn time (*SyT*). The repetition could point to a systemic issue in the process synchronization mechanism, leading to unnecessary delays.
- pt_5^{loan} presents a situation where the synchronization of application cancellations is delayed (*SyT*) due to a high pooling time concerning offers (*PT*). This pattern may imply that cancellations are being processed too slowly due to resources being tied up in offer pooling.
- pt_6^{loan} denotes a scenario where the synchronization of application cancellations (*SyT*) is hampered by a high readiness time regarding applications (*RT*). This pattern suggests that the readiness of applications is causing a slowdown in the cancellation synchronization process.

Next, we derive several actions to address the temporal patterns of operational constraints, i.e., $a_1^{loan}, \ldots, a_7^{loan} \in \mathbb{U}_{action}$. In order to determine the most appropriate actions, we consider the following factors: the selected temporal patterns and their business implications and the potential effectiveness and feasibility of the action within the loan application process. The elicited actions are as follows:

- a_1^{loan} is to send a warning message to the staff member responsible for the *cancel application* task,
- a_2^{loan} is to alter the business rule associated with the *create offer* activity, so that the new rule encourages the acceptance of additional offers by applicants, e.g., by reducing the interest on loans for additional offers,
- a_3^{loan} is to modify the business rule associated with the *call applicants* activity to discourage repeated calls to applicants,
- a_4^{loan} is to revise the business rule associated with the *cancel application* activity to allow for additional resources to carry out the cancellation of applications,
- a_5^{loan} is to adjust the business rule associated with the *cancel application* activity to prevent the cancellation of applications with a single offer,
- a_6^{loan} is to send reminders to applicants about their pending offers, thereby reducing the likelihood of cancellations, and
- a_7^{loan} is to alert loan application managers.

Finally, we design an action graph using the temporal pattern trees and the actions. Fig. 13 shows the action graph.

5.1.2. Generating and planning actions

Using the action graphs, we apply the action engine to the loan application process for 12 weeks from October 2016 to December 2016. In the first week, we analyze the constraint instances from 30.Sep.2016 to 06.Oct.2016 and produce action instances starting from 07.Oct.2016, and, in the second week, we analyze the constraint instances from 07.Oct.2016 to 13.Oct.2016 and produce action instances starting from 14.Oct.2016, etc. We use the action conflict depicted in Fig. 14 to resolve conflicts between actions.

Fig. 15 shows the action plans computed by the action engine. For instance, *action plan 2* in Fig. 15 shows the action plan starting from 14.Oct.2016 and ending at 18.Oct.2016. From 07.Oct.2016 to 13.Oct.2016, pt_1^{loan}, pt_3^{loan}, pt_4^{loan}, and pt_6^{loan} occur, and thus, the action generator generates an action requirement with $(a_1^{loan}, 4)$, $(a_3^{loan}, 2)$, $(a_4^{loan}, 2)$, $(a_6^{loan}, 3)$, and $(a_7^{loan}, 2)$. Action a_1^{loan}, a_2^{loan}, a_5^{loan}, and a_6^{loan} are planned on 14.Oct.2016 and completed on 18.Oct.2016, 16.Oct.2016, 17.Oct.2016, and 15.Oct.2016, respectively. Action a_4^{loan} is planned after a_3^{loan} is completed since a_3^{loan} is conflicting with a_4^{loan}, and a_3^{loan} needs to be executed before a_4^{loan}.

Each action plan involves a different number of actions since the presence of temporal patterns varies in different periods. For instance, *action plan 9*, the action plan starting from 02.Dec.2016, comprises all the actions since all the selected temporal patterns occur in the week starting from 25.Nov.2016 to 01.Dec.2016, which demonstrates the severity of problems in the week. In contrast, *action plan 10*, the action plan starting from 09.Dec.2016, includes two actions, i.e., a_6^{loan}

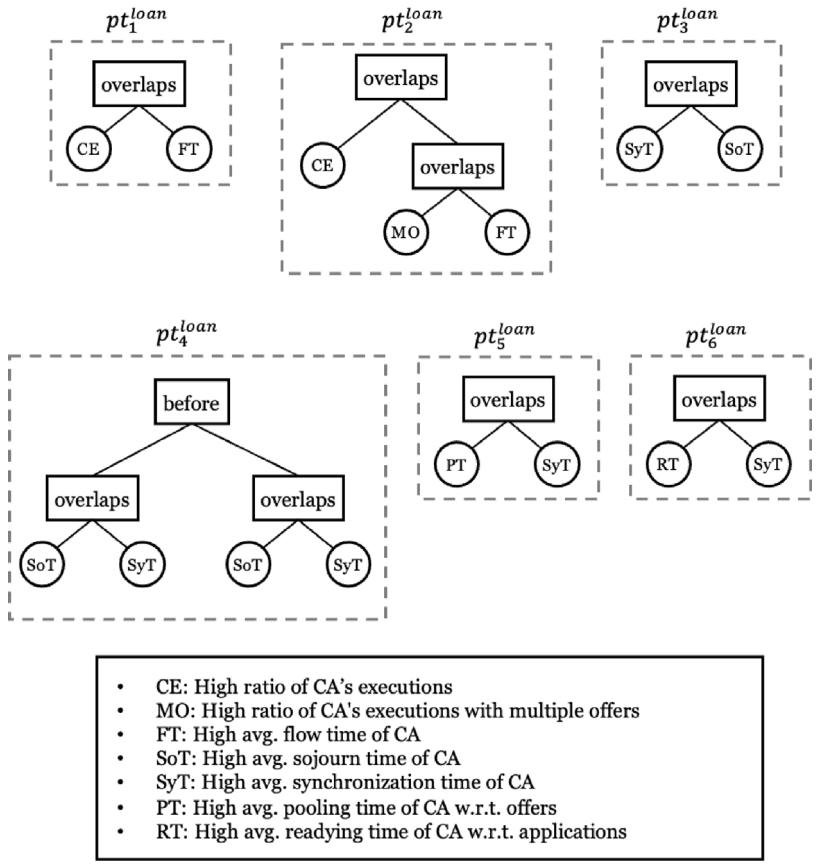


Fig. 12. Temporal pattern trees defined over operational constraints including *CE*, *MO*, *FT*, *SoT*, *SyT*, *PT*, and *RT*.

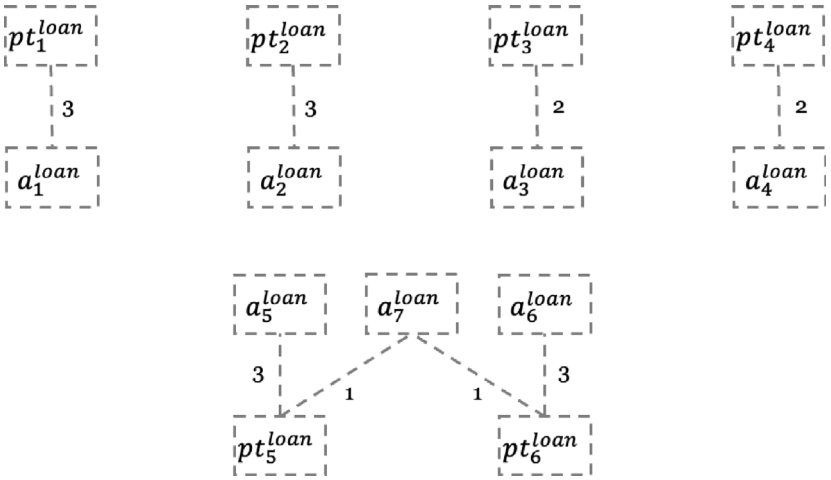


Fig. 13. An action graph that describes the handling of temporal patterns $pt_1^{loan}, \ldots, pt_6^{loan} \in \mathbb{U}_{pt}$ with actions $a_1^{loan}, \ldots, a_7^{loan} \in \mathbb{U}_{action}$.

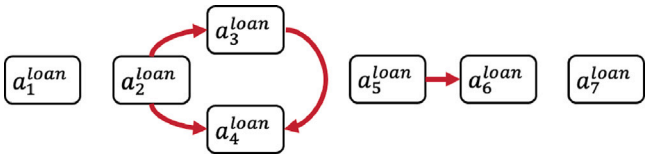


Fig. 14. A conflict graph used to resolve conflicts between actions in the loan application process.

and a_7^{loan}, since only one temporal pattern, i.e., pt_6, is present in the week starting from 02.Dec.2016 to 08.Dec.2016.

We measure several performance metrics for the generated action plans, including makespan, Total Waiting time (TW), and Total Flow time (TF). The waiting time of actions indicates the time taken to start the action, e.g., the waiting time of a_4^{loan} in *action plan 2* is 2 days, whereas the flow time of actions denotes the time taken to complete the action, e.g., the flow time of a_4^{loan} in *action plan 2* is 4 days. For instance, the makespan of *action plan 2* is 4, while the TW and TF are 2 and 14, respectively.

Different action plans show varying total waiting times. For instance, *action plan 9* has a total waiting time of 11, whereas *action*

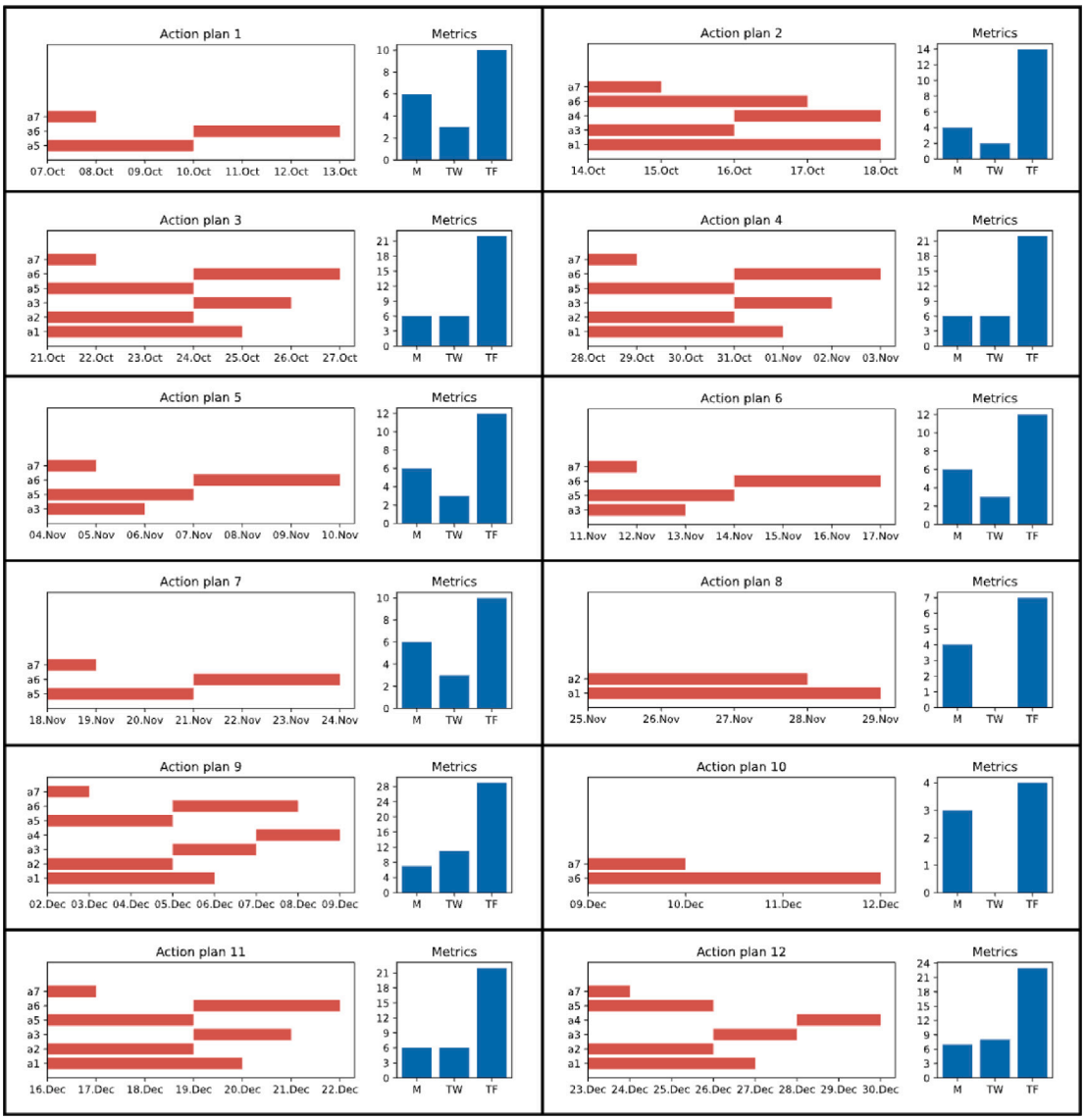


Fig. 15. Action plans computed by the action engine for 12 weeks from October 2016 to December 2016. M, TW, and TF indicate makespan, Total Waiting time, and Total Flow time, respectively.

plan 10 shows no waiting time. Depending on the type of actions, the waiting time could be essential, e.g., the earlier the action is executed, the higher its effect is. The waiting time for an action depends on the existence of its conflicting actions that need to precede the action. For instance, a_6^{loan} often has waiting times, i.e., 8 out of 9 weeks, since it is mostly planned together with a_5^{loan} that is conflicting, precedent action of a_6^{loan}. Moreover, a_4^{loan} always shows waiting times since its preceding action, a_3^{loan}, is always planned together, i.e., 3 out of 3 weeks.

5.2. Experiments

The presented use case demonstrates that different action plans generated by the pattern-based action engine show varying scheduling performance. The scheduling performance may depend on (1) the number of actions in a given action requirement, (2) the number of conflicts between the actions in a given action conflict, and (3) the type of conflicts. To evaluate the scheduling performance of the proposed action engine in a comprehensive manner, we design the following research questions:

- **RQ1**: How do the number of actions in the action requirement and the number of conflicts in the action conflict, affect the computation time and scheduling performance?
- **RQ2**: How do different action conflicts of the same size affect the computation time and scheduling performance of the action planner?

In the following, we explain experiments to answer the aforementioned research questions. The source code and manual for the experiments is publicly available at https://github.com/aopm/pattern-based-action-engine.git.

5.2.1. Experiments regarding RQ1

To address the first research question, we establish a pool of 1000 actions and generate nine distinct action conflict scenarios, each representing a different level of conflict among action pairs. For example, the scenario denoted as "action conflict (10%)" reflects conflicts occurring in 10% of all possible action pairs. With these conflict scenarios, we then generate action plans involving various numbers of actions from different action requirements, ranging from 100 to 1000.

In this experimental setup, we assume each action has a uniform duration of 1. We maintain this constant duration to control for potential variability it may introduce, enabling us to better isolate and understand the effects of our primary variables of interest: the number of actions within an action requirement and the degree of conflict between actions in an action conflict.

In assessing scheduling performance, we utilize makespan as our evaluative measure, primarily due to its direct alignment with the objectives of our proposed action planner. Moreover, makespan offers a valuable means to determine whether necessary actions are being planned and executed quickly for the enhancement of business processes.

Fig. 16(a)–(b) shows the runtime of the action planner when planning a variable number of actions from 100 to 1000 with the nine action conflicts. Each line in Fig. 16(a) represents the runtime of planning the variable number of actions with an action conflict. For instance, the pink-colored line in Fig. 16(a) describes the runtime of planning the variable number of actions with *action conflict 90%*. The experimental results show that the runtime increases quadratically to the number of actions in an action requirement for each action conflict. As the number of actions in an action requirement increases, the runtime difference between different action conflicts increases. For instance, the runtime difference between *action conflict (90%)* and *action conflict (80%)* is around 5 s with 500 actions in an action requirement and around 35 s with 1000 actions in an action requirement.

Each line in Fig. 16(b) describes the runtime of planning the actions of action requirements with different action conflicts. For instance, the yellow-colored line in Fig. 16(b) describes the runtime of planning 1000 actions of an action requirement with different action conflicts. It shows that the runtime increases proportionally to the number of conflicts between actions. When the number of actions in the action requirement is less than or equal to 200, almost no differences exist in the runtime.

Next, Fig. 16(c)–(d) shows the makespan of the action planner when planning a variable number of actions of different action requirements from 100 to 1000 with the nine different action conflicts. Each line in Fig. 16(c) represents the makespan of action plans generated with the variable number of actions and an action conflict of a fixed number of conflicts. For instance, the pink-colored line describes the makespan of action plans generated for the variable number of actions with *action conflict (90%)*. The experimental result shows that the makespan increases linearly to the number of candidates for each action conflict.

As shown in Fig. 16(d), the higher the number of conflicts is, the higher the makespan is. However, the makespan shows more significant increases for specific changes in the number of conflicts. For instance, when action conflict changes from *action conflict (40%)* to *action conflict (50%)*, the makespan increases more than other changes for each number of actions in the action requirement.

5.2.2. Experiments regarding RQ2

The previous experiment considers ten action requirements and nine action conflicts to evaluate the impact of the number of actions and the number of conflicts on the scheduling performance and computation time. In this experiment, we shift our focus to understanding the effects of varying conflict relationships. More specifically, we aim to investigate how different conflict relationships impact the planning process, even when the total number of conflicts remains constant.

To conduct this experiment, we utilize an action requirement involving 500 actions. We subsequently define 99 categories of action conflicts. Each category represents a specific percentage of conflicts among the actions. For example, "action conflict (1%)" symbolizes conflicts in 1% of all possible action pairs, while "action conflict (99%)" represents conflicts in 99% of all action pairs. Within each category, we alter the conflict relationships while keeping the overall number of conflicts constant.

Fig. 17 shows the makespan of the action planner when planning the 500 actions under different categories of action conflicts. Each box plot showcases the makespan when planning the 500 actions with ten unique conflict relationships in the respective category. For instance, the first box plot of *action conflict (1%)* describes the makespans of planning 500 actions with ten different action conflicts with the same total number of conflicts (i.e., 1% of the action pairs have conflicts), but varying conflict relationships. The experimental results show that the variance of makespans continuously increases until a certain point around *action conflict (50%)* and incrementally decreases afterward on a small scale. The variance is significantly small when the number of conflicts is minimal, e.g., 1 and 2 percent of all the action pairs, and the number of conflicts is substantial, e.g., 99 percent of all the action pairs.

Next, Fig. 18 shows the runtime of the same experiment. Each box plot signifies the runtime while planning 500 actions with ten different conflict relationships in the corresponding category. For instance, the first box plot of *action conflict (1%)* describes the runtime of planning 500 actions with ten different action conflicts with the same total number of conflicts (i.e., 1% of the action pairs have conflicts), but varying conflict relationships. Unlike the makespan, the runtime variance incrementally increases as the total number of conflicts increases. However, the runtime does not show significant differences when almost all the action pairs have conflicts, i.e., with *action conflict (99%)*. In some cases, the runtime is hugely higher than the others. For instance, one of the action conflicts in *action conflict (75%)* shows an exceptionally high runtime, even higher than the larger size of action conflicts does, e.g., *action conflict (76%)* and *action conflict (78%)*.

6. Discussion

This section presents a discussion of the evaluation results, limitations, and implications of this work.

6.1. Evaluation results

The use case on the loan application process (cf. Section 5.1) demonstrates that real-life business processes entail temporal patterns of operational constraints, and the proposed pattern-based action engine effectively analyzes such temporal patterns and generates necessary process management actions to deal with the resulting problems. First, a process analyst can flexibly design action graphs using graphical notation. In the use case, we designed an action graph that tackles six temporal patterns of operational constraints with seven actions. Second, constraint instances can be continuously monitored, and actions can be automatically generated based on the designed action graph. In the use case, we monitored the loan application process weekly and automatically generated actions for the coming days. Finally, the proposed action engine's action planner optimizes the generated actions' schedule. In the use case, a conflict-free action plan of the optimal makespan was successfully produced for each week.

As we analyzed the use case, we made noteworthy observations. First, due to uncertain demands for loans and flexible terms for communications, the process showed dynamic behavior each week, resulting in the occurrence of various temporal patterns. Therefore, in some weeks, most of the seven actions were planned, whereas only a few were planned for the other weeks. This, in turn, resulted in varying makespans of the action plans in other weeks. Second, although the action planner is not implemented to optimize the total waiting time and total flow time, all the action plans for the 12 weeks showed the minimum total waiting time and flow time. Third, the waiting time of an action plan results from a specific action in the action requirement. This is because the action is often planned together with its conflicting actions. For instance, a_6^{loan} is often planned together with a_5^{loan}, while a_4^{loan} is always planned together with a_3^{loan}, leading to the delay of a_6^{loan} and a_4^{loan} until a_5^{loan} and a_3^{loan} are completed, respectively.

The experimental results (cf. Section 5.2) demonstrated that the runtime of the proposed action planner is quadratic to the number of

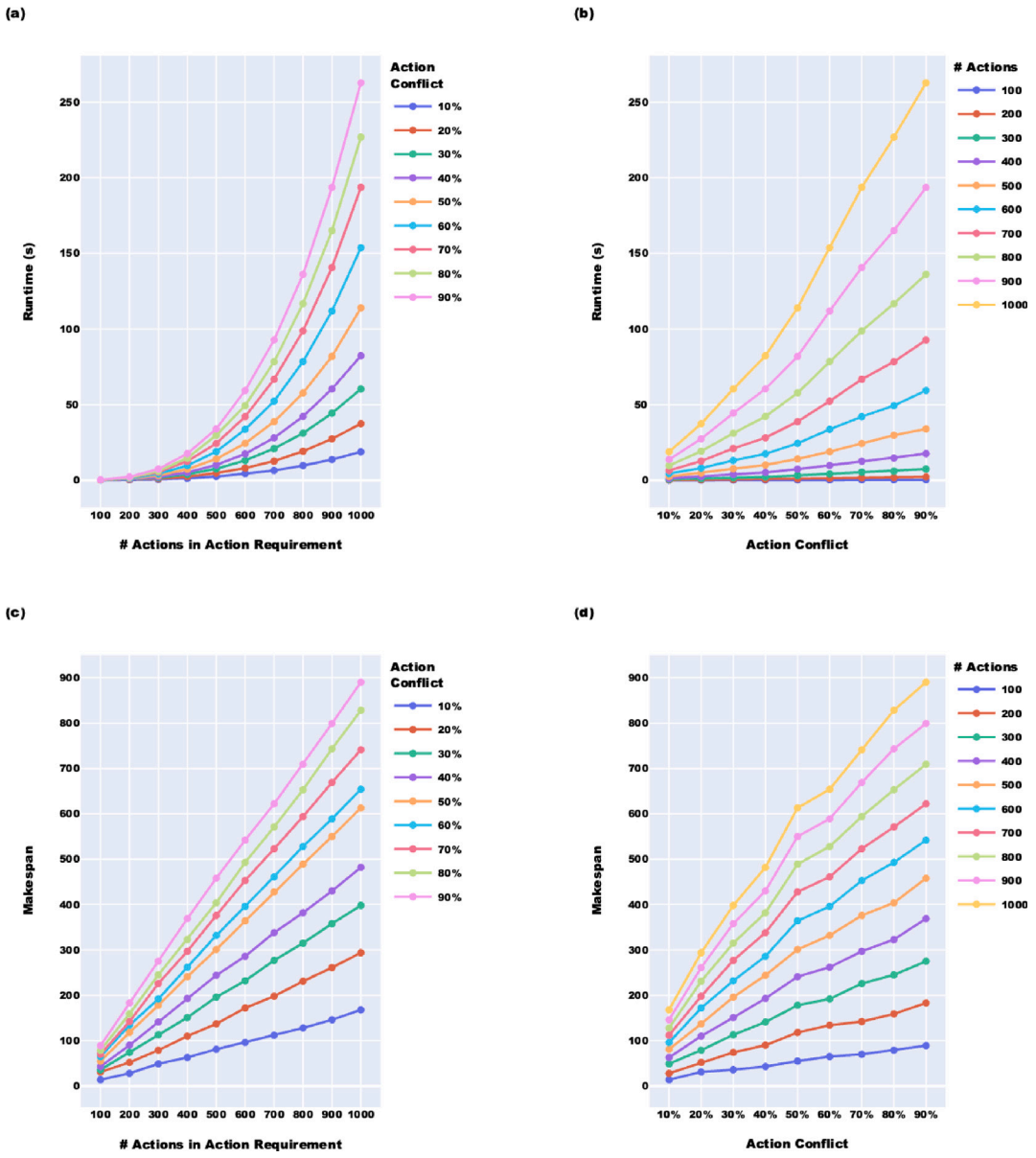


Fig. 16. Experimental results to show the scalability and performance of the proposed action engine.

actions in a given action requirement and proportional to the number of conflicts in a given action conflict. For instance, when the number of actions in the given action requirement is 1000, and 90% of the actions have conflicts, the runtime is 250 seconds. Since the number of process management actions in real-life business processes does not scale to thousands and the automatic generation of such actions does not occur every minute, the proposed approach is scalable to be deployed in real-life business processes.

6.2. Limitations

Although the use case shows the feasibility of the proposed action engine in a real-life business process, it was not conducted in the organization and, thus, lacks the actual application of the action plans to the process. Indeed, it is essential to conduct the case study in the organization to evaluate the effects of actions due to various factors (Dees et al., 2019). For instance, diverse influencing factors may arise while applying the actions, e.g., business processes are affected by social factors such as organizational frictions (de Leoni et al., 2020). Adding a new resource to a task, in principle, increases the productivity of the task. However, considering the possible organizational frictions of the action, it can also negatively impact productivity.

Second, we abstract from defining actions generated by the action engine, leaving it up to domain-specific experts. A recent development in workflow automation systems is a promising solution to translating abstracted actions into formats the source systems can execute. For instance, Make[1] is a web-based automation platform that provides a vast amount of workflow automation scenarios in various information systems, e.g., SAP S/4HANA,[2] ServiceNow,[3] HubSpot,[4] with visualizations for the scenarios. For instance, one can connect SAP S/4HANA to create automated visual workflows, e.g., canceling a supplier invoice, deleting a credit memo request, etc.

Third, the action planner of the proposed action engine resolves the conflicts between the actions of an action requirement. In other words, it does not consider the action instances already being executed

[1] https://www.make.com/
[2] https://www.sap.com
[3] https://www.servicenow.com
[4] https://www.hubspot.com

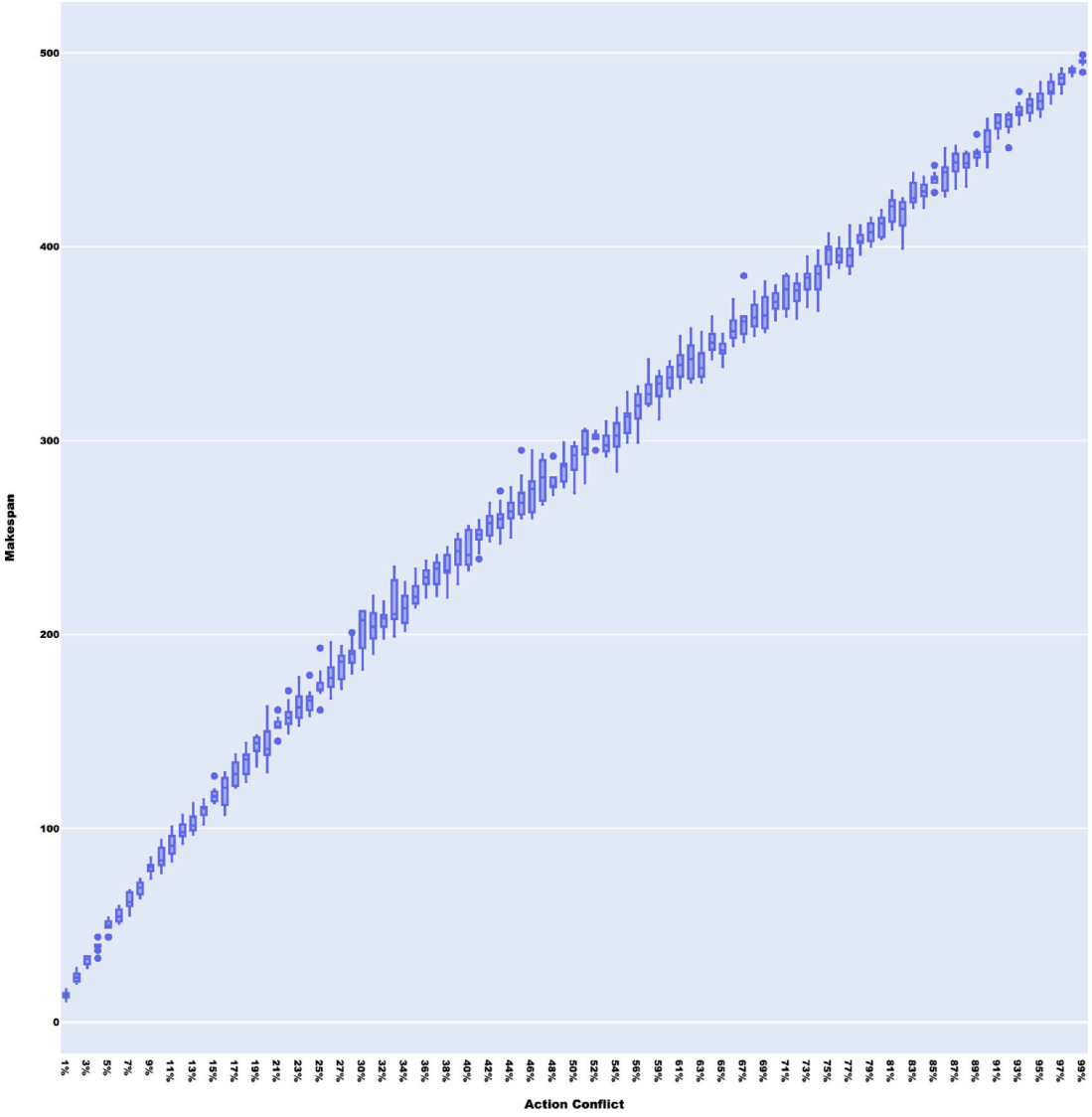


Fig. 17. Experimental results showing the effect of different action conflicts of the same size on the scheduling performance of the proposed action engine.

in the system. For instance, when applied weekly, it analyzes constraint instances in a week, generates actions to tackle the issues in the week, and produces action instances for the coming days. However, if the action instances range longer than a week, the newly generated action instances may conflict with the previous week's action instances. The action planner needs to be extended to consider such ongoing action instances when planning the actions of action requirements.

Lastly, our proposed model produces an action plan for a fixed action requirement. As demonstrated in the evaluation, this approach may result in the start of action execution in the week following the detection of problems. While this might appear delayed, it serves a specific purpose in scenarios that do not mandate immediate intervention and are better addressed through advanced planning and subsequent execution. For instance, situations that necessitate the reassignment of resources or complex logistical coordination stand to benefit from this approach. However, there are situations that demand prompt response, and any delay could result in the issue dissipating before action is taken. To deal with such situations, it is necessary to extend the proposed approach to incorporate planning for "streaming" action requirements, enabling to plan and execute actions as soon as problems arise.

6.3. Implications

Our work has important implications from both academic and practical viewpoints. From an academic research viewpoint, this work provides the foundation for developing novel techniques for analyzing temporal patterns of operational constraints, i.e., operational problems. For instance, one can extend action graphs to restrict the time of evaluating the presence of temporal pattern trees, e.g., if the temporal pattern tree occurs in the last three days of the given constraint instances. Moreover, the generation of action requirements can be enhanced by extending action graphs. For instance, one can add the due date constraint to action graphs, e.g., the corresponding action should be executed in three days.

Furthermore, this work provides the formal architecture upon which novel techniques for generating action requirements and planning action instances can be developed. In this work, we use a brute-force approach to implement the generation of action requirements. Instead, other approaches with different analytical advantages or computational efficiencies can be deployed to implement the action generation. Moreover, our implementation of action planners optimizes the makespan of action plans. Depending on various purposes, one can define different objectives, e.g., optimizing the total waiting time of action plans, and

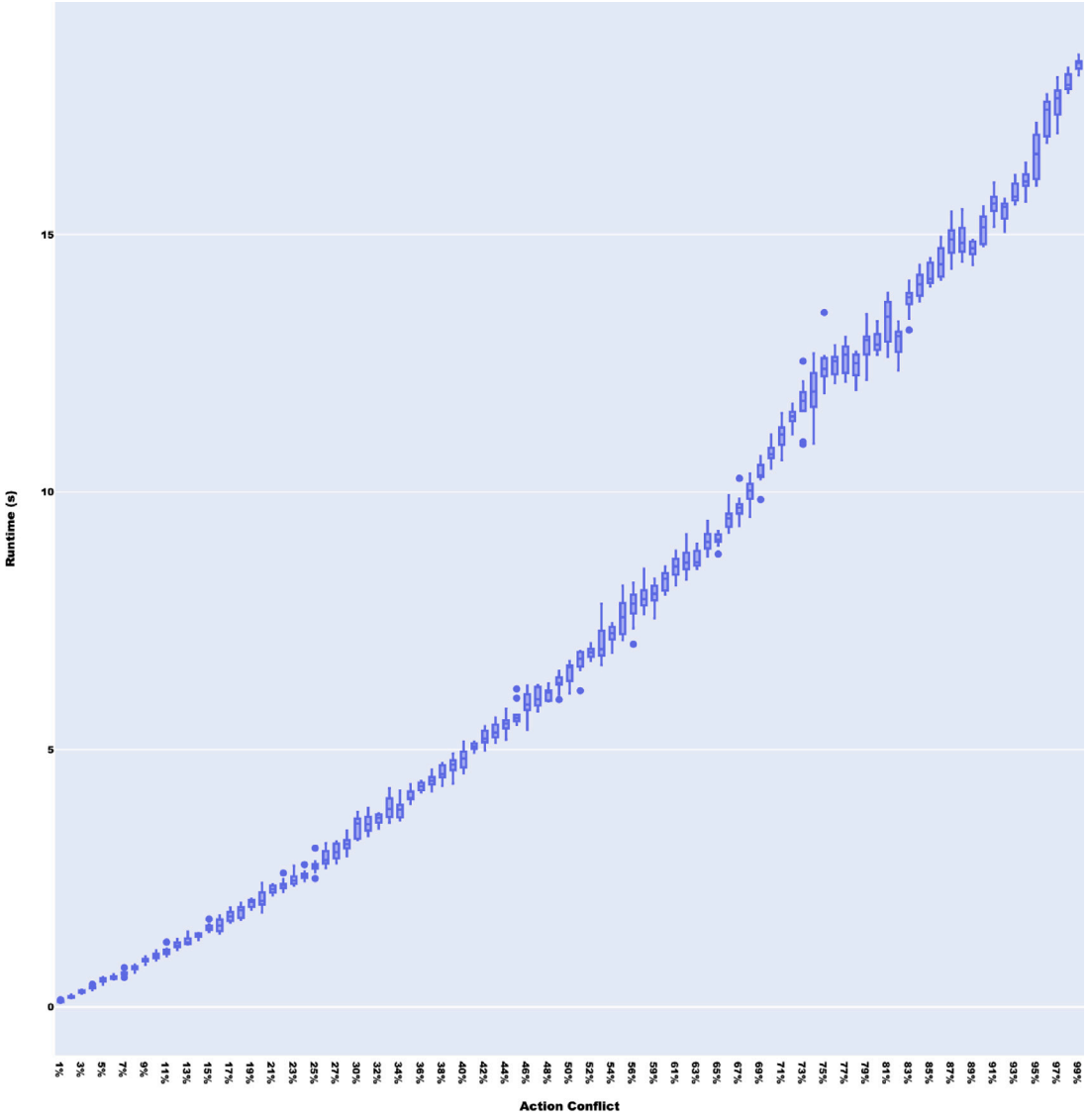


Fig. 18. Experimental results showing the effect of different action conflicts of the same size on the runtime of the proposed action engine.

develop different algorithms to achieve the objectives. In addition, the action planner can be extended to resolve different types of conflicts, e.g., effect-based conflicts and dependency-based conflicts.

Besides, in real-life business processes, it is common to observe variations and changes in process behavior, i.e., concept drifts (Adams et al., 2023). In order to implement effective techniques for action engines, we need to consider such concept drifts. In particular, the design of action graphs should be adapted accordingly to the concept drift so that they remain relevant in varying situations. To this end, opportunities exist for fellow scholars to connect the detection, characterization, and explanation of concept drifts in business processes to the dynamic and adaptive design of action graphs.

Furthermore, the proposed action engine can be extended by distinguishing between observed and predicted constraint instances. The input for the action engine is a collection of constraint instances that may be observed using backward-looking techniques and predicted using forward-looking techniques. For example, an approach suggested in Park and Song (2020) predicts the performance of business processes in a given time window and can be used to produce such predicted constraint instances. The significance of these forecasted operational constraints may vary, potentially influenced by factors like the likelihood or precision of the prediction. To optimize its effectiveness, an action engine can be designed to assign varying weights to constraint instances, thereby prioritizing those constraint patterns bearing higher weights.

From a practical viewpoint, this work allows organizations to continuously analyze temporal patterns of operational constraints detected by various process monitoring techniques and automatically connect the resulting insights into concrete forms of actions. The automated actions tackle the temporal patterns of operational constraints existing in organizations' business processes and contribute to improving the process. This allows organizations to maintain competitive advantages in fast-evolving and dynamic business environments. Moreover, the proposed action engine involves practitioners in designing action graphs. This allows them to leverage domain knowledge to define temporal patterns of operational constraints and elicit actions to deal with the temporal patterns, facilitating the adoption of the technique.

7. Conclusion

Existing approaches in action-oriented process mining consider operational constraints as temporally independent and point-based data, generating actions based on relatively simple occurrence rules (cf. Fig. 1). Moreover, they do not consider possible conflicts between actions that are prevalent in reality when triggering the generated actions. This paper tackles this research gap by proposing a pattern-based action engine to analyze complex temporal patterns of operational constraints and produce conflict-free actions by considering possible conflicts between the actions.

Existing approaches in action-oriented process mining view operational constraints as temporally independent and point-based data and do not consider the potential conflicts between actions. This paper aims to address this research gap by introducing a pattern-based action engine. This action engine is designed to analyze the intricate temporal patterns of operational constraints and produce conflict-free actions, while considering the potential conflicts that might arise between actions.

The proposed action engine consists of three phases. First, domain experts design *action graphs* that are graphical notations consisting of two types of nodes, i.e., temporal pattern trees and actions, and edges connecting the nodes. Using the graphical notations, one can visually define action graphs. Second, the action generator evaluates if the temporal patterns specified in the action graphs occur in the constraint instances of the time window and generates action requirements that describe which actions need to be executed for how long. Finally, the action planner produces an action plan such that no conflicts occur during the execution of the actions.

We demonstrated a use case using the data of a real-life loan application process of a Dutch financial institute to evaluate the feasibility of the proposed action engine in real-life business processes. In the use case, we designed an action graph to tackle the redundant operational costs caused by the cancellation of applications. Afterward, we applied the action engine to the loan application process with the action graphs for 12 weeks. The use case demonstrates that the proposed pattern-based action engine effectively analyzes temporal patterns of operational constraints existing in the real-life business process and automatically generates actions to tackle the risks caused by the patterns. Furthermore, we conducted experiments to evaluate the scalability and performance of the proposed action engine.

In future work, we plan to apply the proposed action engine in real-life business processes. This application will enable us to collaborate with domain experts to derive actual issues present in the process and to exercise full control in addressing these problems with the suggested actions. Subsequently, this application will also allow us to assess the real-world efficacy of the actions generated by our proposed action engine in resolving operational issues. For example, we could measure the impact of these actions by comparing Key Performance Indicators (KPIs) before and after the implementation of the proposed actions. Moreover, we plan to extend the proposed action engine's implementation to generate executable actions by incorporating workflow automation systems. Another interesting direction for future work is to extend the action planner such that the plans are optimized to maximize the impact of actions. To that end, we may predict the impact of generated actions and extend the objective of action planners to maximize the overall impact of the actions.

CRediT authorship contribution statement

Gyunam Park: Conceptualization, Data curation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Daniel Schuster:** Conceptualization, Software, Writing – original draft, Writing – review & editing. **Wil M.P. van der Aalst:** Funding acquisition, Supervision, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We have shared the link to the public dataset.

Acknowledgment

We thank the Alexander von Humboldt (AvH) Stiftung for supporting our research.

References

Adams, J.N., van Zelst, S.J., Rose, T., van der Aalst, W.M.P., 2023. Explainable concept drift in process mining. Inf. Syst. 114, 102177. http://dx.doi.org/10.1016/j.is.2023.102177, URL https://www.sciencedirect.com/science/article/pii/S0306437923000133.

Allen, J.F., 1983. Maintaining knowledge about temporal intervals. Commun. ACM 26 (11), 832–843. http://dx.doi.org/10.1145/182.358434.

Asai, T., Abe, K., Kawasoe, S., Sakamoto, H., Arimura, H., Arikawa, S., 2004. Efficient substructure discovery from large semi-structured data. IEICE Trans. Inf. Syst. 87 (12).

Ayres, J., Flannick, J., Gehrke, J., Yiu, T., 2002. Sequential pattern mining using a bitmap representation. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada. ACM, pp. 429–435. http://dx.doi.org/10.1145/775047.775109.

Badakhshan, P., Bernhart, G., Geyer-Klingeberg, J., Nakladal, J., Schenk, S., Vogelgesang, T., 2019. The action engine – turning process insights into action. In: 2019 ICPM Demo Track. ceur-ws, Aachen, Germany, pp. 28–31, URL http://ceur-ws.org/Vol-2374/paper8.pdf.

Beerepoot, I., Di Ciccio, C., Reijers, H.A., Rinderle-Ma, S., Bandara, W., Burattin, A., Calvanese, D., Chen, T., Cohen, I., Depaire, B., Di Federico, G., Dumas, M., van Dun, C., Fehrer, T., Fischer, D.A., Gal, A., Indulska, M., Isahagian, V., Klinkmüller, C., Kratsch, W., Leopold, H., Van Looy, A., Lopez, H., Lukumbuzya, S., Mendling, J., Meyers, L., Moder, L., Montali, M., Muthusamy, V., Reichert, M., Rizk, Y., Rosemann, M., Röglinger, M., Sadiq, S., Seiger, R., Slaats, T., Simkus, M., Someh, I.A., Weber, B., Weber, I., Weske, M., Zerbato, F., 2023. The biggest business process management problems to solve before we die. Comput. Ind. 146, 103837. http://dx.doi.org/10.1016/j.compind.2022.103837, URL https://www.sciencedirect.com/science/article/pii/S0166361522002330.

Bozorgi, Z.D., Teinemaa, I., Dumas, M., Rosa, M.L., Polyvyanyy, A., 2021. Prescriptive process monitoring for cost-aware cycle time reduction. In: Ciccio, C.D., Francescomarino, C.D., Soffer, P. (Eds.), 3rd International Conference on Process Mining, ICPM 2021, Eindhoven, the Netherlands, October 31 - Nov. 4, 2021. IEEE, pp. 96–103. http://dx.doi.org/10.1109/ICPM53251.2021.9576853.

Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G., 2005. Towards a taxonomy of software change. J. Softw. Maintenance Evol.: Res. Practice 17 (5), 309–332. http://dx.doi.org/10.1002/smr.319, URL https://onlinelibrary.wiley.com/doi/10.1002/smr.319.

Chapin, N., Hale, J.E., Khan, K.M., Ramil, J.F., Tan, W.-G., 2001. Types of software evolution and software maintenance. J. Softw. Maintenance Evol.: Res. Practice 13 (1), 3–30. http://dx.doi.org/10.1002/smr.220, URL https://onlinelibrary.wiley.com/doi/10.1002/smr.220.

Charfi, A., Mezini, M., 2004. Hybrid web service composition: business processes meet business rules. In: Aiello, M., Aoyama, M., Curbera, F., Papazoglou, M.P. (Eds.), Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings. ACM, pp. 30–38. http://dx.doi.org/10.1145/1035167.1035173.

Cheng, T., Sin, C., 1990. A state-of-the-art review of parallel-machine scheduling research. European J. Oper. Res. 47 (3), 271–292. http://dx.doi.org/10.1016/0377-2217(90)90215-W, URL https://www.sciencedirect.com/science/article/pii/037722179090215W.

Chi, Y., Yang, Y., Xia, Y., Muntz, R.R., 2004. Cmtreeminer: Mining both closed and maximal frequent subtrees. In: Advances in Knowledge Discovery and Data Mining: 8th Pacific-Asia Conference, PAKDD 2004, Sydney, Australia, May 26-28, 2004. Proceedings 8. Springer.

Conforti, R., de Leoni, M., Rosa, M.L., van der Aalst, W.M.P., ter Hofstede, A.H.M., 2015. A recommendation system for predicting risks across multiple business process instances. Decis. Support Syst. 69, 1–19. http://dx.doi.org/10.1016/j.dss.2014.10.006.

de Leoni, M., Dees, M., Reulink, L., 2020. Design and evaluation of a process-aware recommender system based on prescriptive analytics. In: van Dongen, B.F., Montali, M., Wynn, M.T. (Eds.), 2nd International Conference on Process Mining, ICPM 2020, Padua, Italy, October 4-9, 2020. IEEE, pp. 9–16. http://dx.doi.org/10.1109/ICPM49681.2020.00013.

Dees, M., de Leoni, M., van der Aalst, W.M.P., Reijers, H.A., 2019. What if process predictions are not followed by good recommendations? In: vom Brocke, J., Mendling, J., Rosemann, M. (Eds.), Proceedings of the Industry Forum at BPM 2019 Co-Located with 17th International Conference on Business Process Management (BPM 2019), Vienna, Austria, September 1-6, 2019. In: CEUR Workshop Proceedings, Vol. 2428, CEUR-WS.org, pp. 61–72, URL http://ceur-ws.org/Vol-2428/paper6.pdf.

Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M. (Eds.), 2005. Process-Aware Information Systems: Bridging People and Software Through Process Technology. Wiley, http://dx.doi.org/10.1002/0471741442.

Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A., 2013. Fundamentals of Business Process Management. Springer, http://dx.doi.org/10.1007/978-3-642-33143-5.

Fahrenkrog-Petersen, S.A., Tax, N., Teinemaa, I., Dumas, M., de Leoni, M., Maggi, F.M., Weidlich, M., 2022. Fire now, fire later: alarm-based systems for prescriptive process monitoring. Knowl. Inf. Syst. 64 (2), 559–587. http://dx.doi.org/10.1007/s10115-021-01633-w.

Fehrer, T., Fischer, D.A., Leemans, S.J.J., Röglinger, M., Wynn, M.T., 2022. An assisted approach to business process redesign. Decis. Support Syst. 156, 113749. http://dx.doi.org/10.1016/j.dss.2022.113749.

Gottschalk, F., 2009. Configurable process models (Ph.D. thesis). In: Beta dissertations, Industrial Engineering and Innovation Sciences, http://dx.doi.org/10.6100/IR653896, Proefschrift..

Grosskopf, A., Decker, G., Weske, M., 2009. The Process: Business Process Modeling using BPMN. Meghan Kiffer Press.

Hammer, M., 1990. Reengineering work: Don't automate, obliterate. Harvard Bus. Rev. 68 (4), 104–112.

Harel, O.D., Moskovitch, R., 2021. Complete closed time intervals-related patterns mining. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, the Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021. AAAI Press, pp. 4098–4105, URL https://ojs.aaai.org/index.php/AAAI/article/view/16531.

Höppner, F., 2001. Learning temporal rules from state sequences. In: IJCAI Workshop on Learning from Temporal and Spatial Data. Vol. 25, Citeseer.

Kam, P., Fu, A.W., 2000. Discovering temporal patterns for interval-based events. In: Kambayashi, Y., Mohania, M.K., Tjoa, A.M. (Eds.), Data Warehousing and Knowledge Discovery, Second International Conference, DaWaK 2000, London, UK, September 4-6, 2000, Proceedings. In: Lecture Notes in Computer Science, Vol. 1874, Springer, pp. 317–326. http://dx.doi.org/10.1007/3-540-44466-1_32.

Kubrak, K., Milani, F., Nolte, A., Dumas, M., 2022. Prescriptive process monitoring: *Quo vadis*? PeerJ Comput. Sci. 8, e1097. http://dx.doi.org/10.7717/peerj-cs.1097.

Mansar, S.L., Reijers, H.A., 2007. Best practices in business process redesign: use and impact. Bus. Process. Manag. J. 13, 193–213.

Martínez-Jurado, P.J., Moyano-Fuentes, J., 2014. Lean management, supply chain management and sustainability: A literature review. J. Clean. Prod. 85, 134–150. http://dx.doi.org/10.1016/j.jclepro.2013.09.042, Special Volume: Making Progress Towards More Sustainable Societies through Lean and Green Initiatives, URL https://www.sciencedirect.com/science/article/pii/S0959652613006550.

Martini, M., Schuster, D., van der Aalst, W.M.P., 2023. Mining frequent infix patterns from concurrency-aware process execution variants. 16 (10), 2666–2678. http://dx.doi.org/10.14778/3603581.3603603.

Moskovitch, R., Shahar, Y., 2015. Fast time intervals mining using the transitivity of temporal relations. Knowl. Inf. Syst. 42 (1), 21–48. http://dx.doi.org/10.1007/s10115-013-0707-x.

Papapetrou, P., Kollios, G., Sclaroff, S., Gunopulos, D., 2009. Mining frequent arrangements of temporal intervals. Knowl. Inf. Syst. 21 (2), 133–171. http://dx.doi.org/10.1007/s10115-009-0196-0.

Park, G., van der Aalst, W.M.P., 2020. A general framework for action-oriented process mining. In: del-Río-Ortega, A., Leopold, H., Santoro, F.M. (Eds.), Business Process Management Workshops - BPM 2020 International Workshops, Seville, Spain. In: Lecture Notes in Business Information Processing, 397, Springer, pp. 206–218. http://dx.doi.org/10.1007/978-3-030-66498-5_16.

Park, G., van der Aalst, W.M.P., 2022. Action-oriented process mining: bridging the gap between insights and actions. Progr. Artif. Intell. 1–22.

Park, G., Song, M., 2020. Predicting performances in business processes using deep neural networks. Decis. Support Syst. 129, http://dx.doi.org/10.1016/j.dss.2019.113191.

Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M., 2004. Mining sequential patterns by pattern-growth: The PrefixSpan approach. IEEE Trans. Knowl. Data Eng. 16 (11), 1424–1440. http://dx.doi.org/10.1109/TKDE.2004.77.

Peleg, M., Somekh, J., Dori, D., 2009. A methodology for eliciting and modeling exceptions. J. Biomed. Inform. 42 (4), 736–747. http://dx.doi.org/10.1016/j.jbi.2009.05.003.

Pinedo, M.L., 2012. Scheduling. Vol. 29, Springer.

Pyzdek, T., Keller, P., 2014. Six Sigma Handbook. McGraw-Hill Education.

Reichert, M., Dadam, P., 1998. ADEPT$_{flex}$-supporting dynamic changes of workflows without losing control. J. Intell. Inf. Syst. 10 (2), 93–129. http://dx.doi.org/10.1023/A:1008604709862.

Reijers, H., Liman Mansar, S., 2005. Best practices in business process redesign: an overview and qualitative evaluation of successful redesign heuristics. Omega 33 (4), 283–306. http://dx.doi.org/10.1016/j.omega.2004.04.012, URL https://www.sciencedirect.com/science/article/pii/S0305048304000854.

Rosa, M.L., Dumas, M., ter Hofstede, A.H.M., 2009. Modeling business process variability for design-time configuration. In: Cardoso, J., van der Aalst, W.M.P. (Eds.), Handbook of Research on Business Process Modeling. IGI Global, pp. 204–228. http://dx.doi.org/10.4018/978-1-60566-288-6.ch009.

Rosemann, M., van der Aalst, W.M.P., 2007. A configurable reference modelling language. Inf. Syst. 32 (1), 1–23. http://dx.doi.org/10.1016/j.is.2005.05.003.

Schonenberg, H., Mans, R., Russell, N., Mulyar, N., van der Aalst, W.M.P., 2008. Process flexibility: A survey of contemporary approaches. In: Dietz, J.L.G., Albani, A., Barjis, J. (Eds.), Advances in Enterprise Engineering I, 4th International Workshop CIAO! and 4th International Workshop EOMAS, Held at CAiSE 2008, Montpellier, France, June 16-17, 2008. Proceedings. In: Lecture Notes in Business Information Processing, Vol. 10, Springer, pp. 16–30. http://dx.doi.org/10.1007/978-3-540-68644-6_2.

Schuster, D., van Zelst, S.J., van der Aalst, W.M.P., 2023. Cortado: A dedicated process mining tool for interactive process discovery. SoftwareX 22, 101373. http://dx.doi.org/10.1016/j.softx.2023.101373.

Ur, B., Ho, M.P.Y., Brawner, S., Lee, J., Mennicken, S., Picard, N., Schulze, D., Littman, M.L., 2016. Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes. In: Kaye, J., Druin, A., Lampe, C., Morris, D., Hourcade, J.P. (Eds.), Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016. ACM, pp. 3227–3231. http://dx.doi.org/10.1145/2858036.2858556.

van der Aalst, W.M.P., 1998. The application of Petri nets to workflow management. J. Circuits Syst. Comput. 8 (1), 21–66. http://dx.doi.org/10.1142/S0218126698000043.

van der Aalst, W.M.P., 2016. Process Mining - Data Science in Action, Second Edition. Springer, http://dx.doi.org/10.1007/978-3-662-49851-4.

van der Aalst, W.M.P., Berti, A., 2020. Discovering object-centric Petri nets. Fundam. Inform. 175 (1–4), 1–40. http://dx.doi.org/10.3233/FI-2020-1946.

van der Aalst, W.M.P., Carmona, J. (Eds.), 2022. Process Mining Handbook. In: Lecture Notes in Business Information Processing, Vol. 448, Springer, http://dx.doi.org/10.1007/978-3-031-08848-3.

van Dongen, B., 2017. BPI challenge 2017. http://dx.doi.org/10.4121/UUID:5F3067DF-F10B-45DA-B98B-86AE4C7A310B, URL https://data.4tu.nl/articles/_/12696884/1.

Vanwersch, R.J.B., Vanderfeesten, I.T.P., Rietzschel, E., Reijers, H.A., 2015. Improving business processes: Does anybody have an idea? In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (Eds.), Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings. In: Lecture Notes in Computer Science, Vol. 9253, Springer, pp. 3–18. http://dx.doi.org/10.1007/978-3-319-23063-4_1.

Villafane, R., Hua, K.A., Tran, D., Maulik, B., 2000. Knowledge discovery from series of interval events. J. Intell. Inf. Syst. 15 (1), 71–89. http://dx.doi.org/10.1023/A:1008781812242.

Weber, B., Reichert, M., Rinderle-Ma, S., 2008. Change patterns and change support features - enhancing flexibility in process-aware information systems. Data Knowl. Eng. 66 (3), 438–466. http://dx.doi.org/10.1016/j.datak.2008.05.001.

Weinzierl, S., Dunzer, S., Zilker, S., Matzner, M., 2020. Prescriptive business process monitoring for recommending next best actions. In: Fahland, D., Ghidini, C., Becker, J., Dumas, M. (Eds.), Business Process Management Forum - BPM Forum 2020, Seville, Spain, September 13-18, 2020, Proceedings. In: Lecture Notes in Business Information Processing, Vol. 392, Springer, pp. 193–209. http://dx.doi.org/10.1007/978-3-030-58638-6_12.

Wermelinger, M., Koutsoukos, G., Lourenço, H., Avillez, R., Gouveia, J., Andrade, L.F., Fiadeiro, J.L., 2003. Enhancing dependability through flexible adaptation to changing requirements. In: de Lemos, R., Gacek, C., Romanovsky, A.B. (Eds.), Architecting Dependable Systems II - [the Book Is a Result of the ICSE 2003 Workshop on Software Architectures for Dependable Systems]. In: Lecture Notes in Computer Science, Vol. 3069, Springer, pp. 3–24. http://dx.doi.org/10.1007/978-3-540-25939-8_1.

Wu, S.-Y., Chen, Y.-L., 2007. Mining nonambiguous temporal patterns for interval-based events. IEEE Trans. Knowl. Data Eng. 19 (6), 742–758. http://dx.doi.org/10.1109/TKDE.2007.190613.