An Alignment-based Multi-Perspective Online Conformance Checking Technique

Zsuzsanna Nagy, Agnes Werner-Stark

University of Pannonia, Faculty of Information Technology,
Department of Electrical Engineering and Information Systems,
Egyetem u. 10, 8200 Veszprém, Hungary
nagy.zsuzsanna@virt.uni-pannon.hu, werner.agnes@virt.uni-pannon.hu

Abstract: Conformance checking is a class of process mining techniques, which contrasts the modeled behaviors with the observed behaviors of a process, to detect, locate and explain the deviations between them. Even though deviations can occur anytime and in any perspectives, currently there is no such conformance checking technique available, which is able to take into account other perspectives than the control-flow perspective of the investigated process when computing the conformance statistics on running, incomplete cases. In this paper, a multi-perspective online conformance checking technique is introduced, which aims to confront the modeled behaviors in form of a Data Petri net process model with a stream of events as the observed behaviors. For the conformance checking, two existing techniques were merged: a prefix-alignment-based technique, which is able to compute conformance statistics for incomplete process executions by applying incremental heuristic search, and an alignment-based multi-perspective conformance checking technique, which is able to compute conformance statistics for complete process instances while focusing on multiple perspectives.

Keywords: process mining; conformance checking; multi-perspective; prefix-alignment; incremental heuristic search; event stream; Data Petri net

1 Introduction

In today's information systems, a large amount of data (called *event data*) is generated from the executed business processes. Process mining aims to improve real processes by extracting knowledge from their previously recorded behavior utilizing the available event data. There are three main areas within process mining: process discovery, conformance checking and enhancement [1]. This paper only focuses on conformance checking.

In *conformance checking* [2], the event data is confronted with a (hand-made or discovered) process model of the same process, so the deviations between the observed and the modeled behaviors can be detected, located and explained.

Deviations identified using conformance checking may, for example, in an automated manufacturing process, point at a production tool that sometimes malfunctions (i.e., the tool does not work the same way as it is expected).

In the past years, a great variety of conformance checking techniques were developed, however, there are still not any solutions, which allow conformance analysis online while taking into account multiple perspectives. Existing works in online conformance checking [15-19] focus only on the control-flow perspective and the works in multi-perspective conformance checking [6-8] only allow a posteriori analysis (i.e., the non-conformant behaviors are detected only after the completion of the case). As it was highlighted in [3] too, the vast majority of the existing conformance checking techniques take into account only the control-flow perspective, even though taking into account other perspectives (e.g., time, resource, or data) would be important, because deviations may not only be in the ordering of the activities. For instance, in the case of a manufacturing process, knowing which operation is executed is not enough to tell whether it was executed correctly, knowing the details of the execution (e.g., used resource) is necessary, too. For this reason, a multi-perspective view of such processes is needed. In addition, by enabling it in an online setting, deviations can be detected (in any perspective) in the processes' executions as soon as they occur, so countermeasures can immediately be initiated to reduce the possible negative effects caused by them.

Nowadays, the de facto standard technique for calculating conformance checking statistics is the calculation of *alignments* [11], which provides a "closest path" for each completed process instance through the process model [2]. The differences pinpointed by the alignments can be interpreted by experts, so conclusions can be drawn and actions can be made to improve future process executions.

In recent years, alignment-based online conformance checking techniques were developed [17, 18], which can be applied to ongoing, incomplete cases as well. The difference between the alignment calculation in online and offline settings is that in an online setting the incompleteness of a trace is not acknowledged as incorrect behavior, because the case can still be ongoing. For this reason, in an online setting *prefix-alignments* are computed.

In our previous work [25], a rudimentary solution for the multi-perspective online conformance checking problem was proposed, which was an extended version of the latest online conformance checking solution at the time, the behavior pattern-based online conformance checking technique [16]. Since then a new, a prefix-alignment-based online conformance checking technique was developed] 18[, which is an exact solution for the online conformance checking problem. In [20], it was proven the most accurate and fastest solution among all the available online conformance checking approaches.

In this paper, an exact solution for the multi-perspective online conformance checking problem is proposed. The presented method returns an optimal multi-perspective prefix-alignment between a multi-perspective process model and an

event stream. This method is a merge of two existing conformance checking techniques: the balanced multi-perspective conformance checking (BMCC) [8] and the prefix-alignment based online conformance checking (OCC) [18] technique. The solution was tested on real-life processes, including a manufacturing process.

The proposed solution could be used to monitor the executions of processes with a short lead time and a prescriptive process model. It could realize a low-cost temporary monitoring system for immature, not yet well-developed processes until the process reaches a well-developed stage. It could also support old processes, where the monitoring system is not able to detect all anomalies and the system cannot be modified (e.g., the manufacturing process examined in Section 4 of this paper). It could be used as a supplementary solution for detecting deviations.

The remainder of this paper is structured as follows. In Section 2, the basic concepts are presented to get a better understanding of the proposed technique. In Section 3, the proposed technique is described in detail, then in Section 4, it is evaluated and the results of the experiments are presented. Finally, the paper is concluded.

2 Background

This section introduces basic concepts related to the proposed technique to help to understand how it works. The concepts are only introduced briefly, for a more detailed description the corresponding works are referred to.

2.1 Event Stream

Most process mining algorithms expect an event log as the input for the observed behavior, which contains a finite number of *events*. Every event must have a *case identifier* and an *activity* recorded. The case identifier identifies the context in which the activity was executed. Real-life events may contain additional information as well, for instance, about the person or machine who executed the activity (i.e., resource) or the time when the activity was executed (i.e., timestamp). A trace is a sequence of events that was executed in the same context.

In this paper, an *event stream* is assumed instead of an event log. The stream consists of *observable units* from which related process information can be extracted. One observable unit corresponds to one event.

Generally, an event stream is defined as an (infinite) sequence of events, where an event is a tuple of a case identifier and an activity identifier [18]. If other perspectives are taken into account as well, then the additional information, the values of the event attributes can be included as a tuple within the event tuple.

Let $\mathcal{C}$ denote the universe of case identifiers, $\mathcal{A}$ denote the universe of activities and $\mathcal{U}$ denote the universe of values. Let $\mathcal{U}^n$ denote the universe of values of n event

attributes in such a way, that $\mathcal{U}^n = \prod_{i=1}^{n} \mathcal{U}_i$ is a Cartesian product, where $\mathcal{U}_i \in \mathbb{P}(\mathcal{U})$ is the universe of values of the i-th event attribute from n event attributes. Therefore, $u^n \in \mathcal{U}^n$ is an n-tuple, where $u^n = (u_1, \dots, u_n) \in \mathcal{U}_1 \times \dots \times \mathcal{U}_n$. An *event*, an *observable unit* $e = (c, a, u^n) \in \mathcal{C} \times \mathcal{A} \times \mathcal{U}^n$ is a trio that describes an event as the execution of activity a with attribute values u^n observed in the context of a process instance identified by a case identifier c. The universe of all possible observable units is defined as $\mathcal{E} = \mathcal{C} \times \mathcal{A} \times \mathcal{U}^n$. An *event stream* is defined as an infinite sequence of observable units: $S \in (\mathcal{C} \times \mathcal{A} \times \mathcal{U}^n)^*$. The definitions of event and event stream are based on the control-flow version of the definition of "Event; Event Stream" in [18].

To process the event data easily, the observed units are logged separately in traces, based on but without the values of their case identifier. Therefore, a *log step* s_l is recorded for each observed event, which is a tuple of an activity a with attribute values u^n (i.e., $s_l = (a, u^n)$). The log steps of the same case are collected into one *trace* σ, so σ is a sequence of log steps recorded for the same case (i.e., $\sigma = \langle s_{l_1}, \dots, s_{l_m} \rangle$, where m is the number of observed events for a case c). Furthermore, in order to obtain the observed value of the i^{th} attribute, a *projection function* π_i is used on the tuple of attribute values u^n. For instance, with π_3 the value of the third attribute can be extracted from u^n (i.e., $\pi_3(u^n) = u_3$). This function can be used on any other tuples and sequences of tuples. For instance, π_1^* can be used on a trace σ to extract the activities from it (i.e., $\pi_1^*(\sigma) = \langle a_1, \dots, a_m \rangle$).

2.2 Data Petri Net (DPN)

The formalism of *Data Petri net (DPN)* or Petri net with data was introduced in [4] and was later revisited in [5].

DPNs are special types of Petri nets that are used to capture the interactions of the control-flow perspective with the other perspectives. It stores the data values in globally defined *variables*, which can be updated by specified transitions through *write operations*. Before the transition fires, the new data values are temporarily stored in *prime variables*. In a DPN, transitions can have data-dependent *guards*. A transition with a guard can fire only if its guard is satisfied. The guard is evaluated based on the linear *guard expression* assigned to it, which can be any formula over the process variables using relational and logical operators. A DPN may contain *invisible transitions* as well, which only appear in the model.

The exact definitions related to DPNs can be found in [9] with examples. In this paper, the notation of some components was changed. Therefore, a DPN is defined by $N = (P, T, F, V_P, V_{dom}, V_{in}, T_{wr}, T_{gd})$, where P, T and F define a simple Petri net, V_P the name of the variables, V_{dom} the domain of the variables, V_{in} the initial value of the variables, T_{wr} the write operations (i.e., for each transition the list of variables it writes) and T_{gd} the guard function for each transition.

A *process step* s_p is a valid firing of a DPN, which is a tuple of the fired transition t and the variable assignments w of that transition (i.e., $s_p = (t, w)$). From valid firings of a DPN, starting from the initial state and finishing in one of the reachable states (not necessarily in one of the final states), a *firing sequence* ρ can be defined (i.e., $\rho = \langle s_{p_1}, s_{p_2}, \dots, s_{p_m} \rangle$, where m is the number of fired transitions).

2.3 Alignments

Alignments were introduced in [10], then were improved in [11]. It was shown in [12], that the performance of alignment computation greatly depends on the adequate parametrization of the underlying search algorithm and in [13] recommendations for parameter configurations were made too, to increase the computation efficiency. In [14] light and efficient methods were introduced for offline alignment computation. The online equivalent of alignment computation, prefix-alignment computation, was introduced in [18].

(Prefix-)alignments are used to explain traces with respect to a reference process model, so they map a trace onto an execution sequence of the process model. The first row of an alignment represents a trace (i.e., a sequence of log steps) and the second row represents a firing sequence of a process model (i.e., a sequence of process steps).

To match the events with the transitions, an *activity label function* $\lambda_{act} : T \to \mathcal{A} \cup \{\tau\}$ is used, which maps each transition of a (Data) Petri net to an observable activity name or τ (if the transition is invisible). Similarly, a *variable label function* $\lambda_{var} : V_P \to \mathbb{N}^+_{\leq n}$ is used to match each process variable to an observable attribute. Assuming the event data comes from an event stream, each variable is mapped to the position, where the value of the corresponding attribute is located within the *n*-tuple of attributes u^n of an observable unit.

When only the control-flow perspective is taken into account, then there can be three types of moves in an alignment: synchronous move, log move and model move. When the other perspectives are taken into account as well, then there can be two kinds of synchronous moves: correct and incorrect synchronous move. A *log move* $(\gg, s_l)$ indicates an unexpected behavior; an activity, which should not have been executed, but was observed (i.e., a log step without a process step). A *model move* $(s_p, \gg)$ indicates a missing behavior; an activity, which should have been executed, but was not observed (i.e., a process step without a log step). A *synchronous move* (s_l, s_p) indicates an expected behavior; an expected activity was executed and was observed (i.e., a log step $s_l = (a, u^n)$ with a process step $s_p = (t, w)$, where $a = \lambda_{act}(t)$). A *correct synchronous move* is a synchronous move, where the expected values of the event attributes were recorded in the event log (i.e., $\forall_{v \in T_{wr}(t)}\, \pi_{\lambda_{var}(v)}(u^n) = w(v)$). In contrast, an *incorrect synchronous move* is a synchronous move, where at least one unexpected value was recorded (i.e., $\exists_{v \in T_{wr}(t)}\, \pi_{\lambda_{var}(v)}(u^n) \neq w(v)$). An alignment move with neither a log step

and a process step (i.e., $(\gg, \gg)$) is an illegal move. Therefore, the set of all legal *multi-perspective alignment moves* can be defined as $\Gamma = (LS \cup \{\gg\}) \times (PS \cup \{\gg\}) \setminus \{(\gg, \gg)\}$, where LS is the set of all the possible log steps and PS is the set of all the possible process steps. A *multi-perspective alignment* $\gamma \in \Gamma^*$ is a finite sequence of legal alignment moves, which is constructed from a trace σ and a firing sequence of a DPN ρ with the help of the labelling functions λ_{act} and λ_{var}.

A multi-perspective alignment move can be transformed into a control-flow alignment move (with information loss) by omitting the attribute values u^n from the log step and the variable assignments w from the process step. Let Γ_c be the set of all legal control-flow alignment moves, then a function $f_c: \Gamma \to \Gamma_c$ can be defined, which gives back the control-flow version of the given multi-perspective alignment move. To transform a multi-perspective alignment $\gamma \in \Gamma^*$ into a control-flow alignment $\gamma_c \in \Gamma_c^*$, a function $f_c^*: \Gamma^* \to \Gamma_c^*$ can be defined.

The goal of alignment-based conformance checking is to find an optimal alignment that minimizes the deviations between a trace and a firing sequence. Deviations are scored according to a *cost function* κ, which can be standard or user − defined. The standard cost function assigns the same cost of 1 to each deviation. At multi-perspective alignments, for each incorrect or missing variable assignment, a cost of 1 is also added.

2.6 Optimal Variable Assignment (OVA) Problem

A more detailed description of the OVA problem can be read in [9].

The OVA problem can be transformed into a Mixed-integer linear programming (MILP) [23, 24] problem, where the constraints are built from the writing operations and the guard functions of the transitions (which are included in the alignment). The goal of the MILP problem is to minimize the cost of deviations between the attribute assignments of the events and the process variable assignments required by the DPN. Therefore, the objective function is the sum of wrong process variable assignments and missing process variable assignments.

2.4 Synchronous Product Net (SPN)

The problem of finding an optimal alignment can be reduced to the shortest path problem. A possible way of computing an optimal (prefix-)alignment is to find the shortest path in the state-space of the SPN [11].

An SPN is a composition of a trace net (a Petri net constructed from the control-flow part of a trace) and a process net, in which each transition corresponds to an alignment move. Hence, any sequence of transitions corresponds to a (prefix-) alignment. The search is guided by the costs which are assigned to each transition according to the given cost function. To compute an optimal (prefix-)alignment, the shortest path (i.e., a sequence of alignment moves with minimal total cost) is looked

for from the initial marking of the SPN to a marking where the last place of the trace net part of the SPN is marked.

The state-space of an SPN can be used for finding an optimal multi-perspective (prefix-)alignment as well. However, the combination of possible variable assignments can be infinite, so to keep the search space minimal, the control-flow copy of the DPN model (i.e., Petri net model) is preferred to be used to build the SPN. The OVAs for an alignment move can depend on both the chosen predecessor and successor moves, so they need to be recomputed whenever the path changes.

2.5 Incremental A* Algorithm

The incremental A* algorithm introduced in [18] is a modified version of the well-known A* algorithm [21]. A* algorithm is suited for computing alignments on small process models. (For large models, the symbolic technique was proven to be more suitable [22].)

The incremental A* algorithm is an informed search algorithm that computes the shortest path for a fixed initial and changing final state in the incremental steps. It can be used to find an optimal prefix-alignment for an incomplete trace from an event stream. The search space is the state-space of the previously discussed SPN. Initially, it only contains the model moves. It is extended with log moves and synchronous moves as new events are observed.

The brief steps of the incremental A* algorithm based on the description in [20] are the followings: (1) A new event $e = (c, b)$ is observed. (2) The SPN is extended for process instance c by the new activity b. For example, assume that the event (c, a) was previously received. When extending an SPN by a new activity, new transitions are added representing a log move on the new activity and potential synchronous moves. (3) The search is continued for the shortest path on the state-space of the extended SPN from previously cached intermediate search-results, i.e., states already explored/investigated. (4) The new prefix-alignment is returned and the search-results are cached.

3 Proposed Technique

In this section, the proposed MOCC technique is introduced. First, an overview of the whole proposed approach is presented, then its components are described in more detail.

3.1 Overview

The proposed technique is a merge of [8] and [18]. Given a DPN process model, a stream of events and a user-defined cost function, it calculates an optimal multi-perspective prefix-alignment each time a new event is observed for a case.

Just like in [18], the core idea of the algorithm is to utilize the previously calculated results of exploring the state-space of an SPN. For each process instance (case), a trace and an SPN are maintained. The trace consists of the events observed so far and the SPN is constructed from the control-flow copy of the process model and the trace. The trace needs to be stored since the SPN only contains the activity names.

When a new event (c, a, u^n) is observed, the trace is extended by this new event and the SPN is extended by activity a (i.e., the activity identifier of the event). Next, the search is continued with the modified incremental A* algorithm by using the pre-filled open and closed sets from the previous search. For each state in the open set, the possible successor states are examined: the control-flow alignment is reconstructed and extended with the possible successor alignment moves, then OVAs are calculated to augment the extended control-flow alignments with them. To calculate the OVAs, the relevant variable writing operations and guard functions of the DPN process model are needed. From the successor states, the one with OVAs and minimal total cost (i.e., minimal alignment and variable cost) is chosen. This way an optimal multi-perspective (prefix-)alignment can be obtained.

To decrease the calculation time, the algorithm can be extended with two things:

1) *Direct Synchronizing*: Similar to the direct synchronizing introduced in [20], the idea is to skip shortest path searches in cases where the previously calculated prefix-alignment can simply be extended by a synchronous move that includes the newly observed event. However, in the multi-perspective case, the shortest path search can be skipped only if the synchronous move is correct (i.e., all variable writing is correct and the guard of the involved transition is satisfied). If direct synchronizing is possible, the heuristic recalculations (which involve a high calculation effort) for finding the shortest path can be avoided, thus speeding up the prefix-alignment computation.

2) *Caching OVAs*: Calculating the OVAs is the most time-consuming part of the whole algorithm, so it is critical to decrease the time spent on it in terms of the applicability of the technique in an online setting. Assuming that the set of possible variable assignment problems is finite (i.e., not every case has an unique trace), by caching the problem and solution pairs, if a problem occurs repeatedly, then the result can be fetched from the cache and the calculations for solving the OVA problem can be skipped. The worst-case complexity of solving the OVA problem is exponential in the number of written variables and guards associated to transitions. However, with a good OVA cache, the worst-case computational complexity is linear in the size of the cache.

The overview of the proposed MOCC approach can be seen in Figure 1.

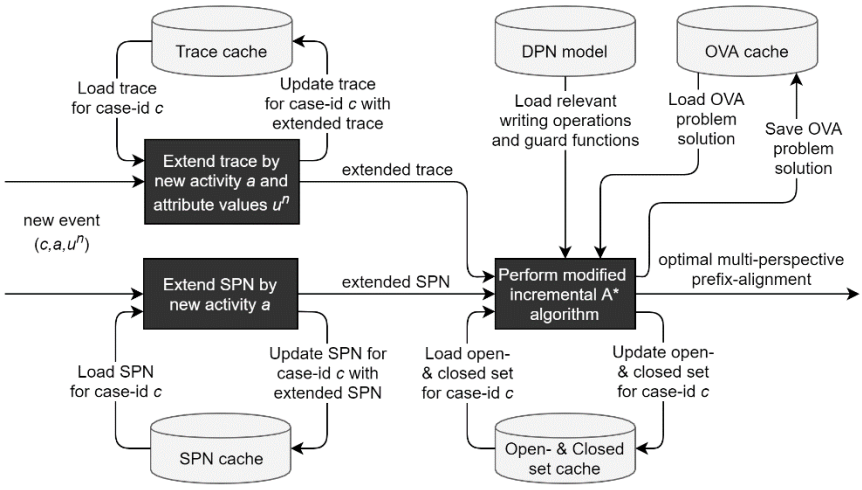


Figure 1
Overview of the proposed MOCC approach

3.2 The Main Algorithm

In this section, the overall algorithm (*Figure 2*) is described.

As input, a labeled DPN reference process model LN, an event stream S, a user-defined cost function κ, a user-defined variable cost function κ_{var}, and a cache that stores the OVAs from previous runs Ω are expected. A labeled DPN consists of a DPN with unique initial marking $[p_i]$ and final marking $[p_o]$, an activity label function λ_{act} and a variable label function λ_{var}. Giving the cost functions are optional; if they are not given, then the standard cost functions will be used (lines 2-5). κ is used for computing the cost of control-flow alignments and also for calculating the heuristics (i.e., the estimated cost to reach the final marking $[p_o]$). κ_{var} is used for computing the cost of the variable assignments and also for calculating the OVAs. It is important, that κ and κ_{var} are defined separately and the sum of them gives the cost of a multi-perspective prefix-alignment. Giving a cache for OVAs Ω is also optional; if it is not given, then a new, initially empty cache will be used (lines 6-7). The algorithm processes every event on the stream S in the order in which they are observed. Since S is an infinite sequence, an infinite cycle is used to iterate in it (line 9).

First, the case identifier c, the activity identifier a and the attribute values u^n are extracted from the currently processed event (observable unit) e (lines 10-13). Next, if it is the first observed event of the case c, then a new SPN is constructed from the activity a and the control-flow part of the DPN process model N. Otherwise, the previously constructed SPN will be extended with the activity a (line 17). For the

state-space of the SPN, a heuristic function h is defined (line 18). If it is the first observed event of the case (i.e., the SPN just have been constructed), then the open set O, the cost-so-far function g, the predecessor function p, the alignment function γ and the current variable values function α will be initialized for the SPN (lines 19-26). Afterward, an optimal multi-perspective prefix-alignment is calculated for case c by calling the modified incremental A* algorithm (line 27). By applying the algorithm, an optimal multi-perspective prefix-alignment $\bar{\gamma}$, the components of the search process (the alignment function γ, the current variable values function α, the open set O, the closed set C, the cost-so-far function g and the predecessor function p) and the updated cache for OVAs Ω are obtained. The function γ, α, g and p assign to already discovered states a multi-perspective prefix-alignment with optimal variable assignments, the current variable values in that state (based on γ), the currently known cheapest cost that leads to that state (the cost of γ) and the corresponding predecessor state, respectively. The results are cached so they can be reused when computing the next optimal multi-perspective prefix-alignment, when a new event is observed for the given case. Afterward, the next event is processed.

Algorithm 1: Incremental Multi-Perspective Prefix-Alignment Computation

input : $LN = ((P, T, F, V, V_{dom}, V_{in}, T_{wr}, T_{gd}), [p_i], [p_o], \lambda_{act}, \lambda_{var}), \mathcal{S} \in (\mathcal{C} \times \mathcal{A} \times \mathcal{U}^n)^\star$,
$\kappa : \Gamma_c \to \mathbb{R}_{\geq 0}, \kappa_{var} : V \times \mathcal{U} \times V_{dom} \to \mathbb{R}_{\geq 0}$,
$\Omega : T^\star \times (\mathbb{N}^+ \times \mathcal{U}^P_{V_{dom}}) \to (\mathbb{N}^+ \times \mathcal{U}^P_{V_{dom}})$

1 **begin**
2 **if** κ *is not defined* **then**
3 let $\kappa : \Gamma_c \to \mathbb{R}_{\geq 0}$; // define standard cost function κ for control-flow alignments
4 **if** κ_{var} *is not defined* **then**
5 let $\kappa_{var} : V \times \mathcal{U} \times V_{dom} \to \mathbb{R}_{\geq 0}$; // define standard variable cost function κ_{var}
6 **if** Ω *is not defined* **then**
7 let $\Omega : T^\star \times (\mathbb{N}^+ \times \mathcal{U}^P_{V_{dom}}) \to (\mathbb{N}^+ \times \mathcal{U}^P_{V_{dom}})$; // define Ω
8 $i \leftarrow 1$;
9 **while** $true$ **do**
10 $e \leftarrow S(i)$; // get i-th event of the event stream
11 $c \leftarrow \pi_1(e)$; // extract case-id from current event
12 $a \leftarrow \pi_2(e)$; // extract activity from current event
13 $u^n \leftarrow \pi_3(e)$; // extract attribute values from current event
14 **if** $c \notin dom(\mathcal{D}_o) \wedge c \in \mathcal{C}$ **then** // initialize cache for new case c
15 $\mathcal{D}_\sigma(c) \leftarrow \langle\rangle, \mathcal{D}_C(c) \leftarrow \emptyset$;
16 $\mathcal{D}_\sigma(c) \leftarrow \mathcal{D}_\sigma(c) \cdot \langle(a, u^n)\rangle$; // extend trace for case c
17 let $N^S = (P^S, T^S, F^S, M_i^S, M_f^S, \lambda_{act}^S)$ from LN and $\pi_1^\star(\mathcal{D}_\sigma(c))$; // construct/extend SPN
18 let $h : \mathcal{R}(N^S, M_i^S) \leftarrow \mathbb{R}_{\geq 0}$; // define heuristic function
19 **if** $|\mathcal{D}_\sigma(c)| = 1$ **then** // initialization for first run regarding case c
20 $\mathcal{D}_O(c) \leftarrow \{M_i^S\}$; // initialize open set
21 $\mathcal{D}_g(c) \leftarrow M_i^S \mapsto 0$; // initialize cost-so-far function
22 $\mathcal{D}_p(c) \leftarrow M_i^S \mapsto (null, null)$; // initialize predecessor function
23 $\mathcal{D}_\gamma(c) \leftarrow M_i^S \mapsto \langle\rangle$; // initialize alignment function
24 $\mathcal{D}_\alpha(c) \leftarrow M_i^S \mapsto \{\}$; // initialize variable values function
25 **forall** $v \in V$ **do** // set initial values for all the variables
26 $\mathcal{D}_\alpha(c) \leftarrow M_i^S \mapsto \mathcal{D}_\alpha(c)(M_i^S) \cup \{v \mapsto V_{in}(v)\}$
27 $\mathcal{D}_{\bar{\gamma}}(c), \mathcal{D}_\gamma(c), \mathcal{D}_\alpha(c), \mathcal{D}_O(c), \mathcal{D}_C(c), \mathcal{D}_g(c), \mathcal{D}_p(c), \Omega \leftarrow$
$A^\star_{inc}(LN, \mathcal{D}_\sigma(c), a, u^n, N^S, \mathcal{D}_O(c), \mathcal{D}_C(c), \mathcal{D}_g(c), \mathcal{D}_p(c), h, \mathcal{D}_\alpha(c), \mathcal{D}_\gamma(c), \kappa, \kappa_{var}, \Omega)$
// execute/continue shortest path search
28 $i \leftarrow i + 1$;

Figure 2
Algorithm of the Incremental Multi-perspective Prefix-Alignment Computation

3.3 Performing the Modified Incremental A* Algorithm

In this section, the algorithm to compute multi-perspective prefix-alignments on the basis of previously executed instances of the modified incremental A* algorithm is presented.

Similar way as in [18], the main idea of this approach is to continue the search on an extended search space as a new event (c, a, u^n) is received. The algorithm is applied using the cached open- and closed-set for case identifier c and the cost function κ on the corresponding extended SPN to grab the control-flow perspective. To be able to take into account other perspectives too, the extended trace σ, the variable cost function κ_{var} and components of the DPN process model are used. Furthermore, to shorten the calculation time, the currently known optimal multi-perspective prefix-alignment γ and variable assignments α are cached for each state in each case and the OVA cache Ω is used too.

In *Figure 3* the algorithmic description of the modified incremental A* approach is presented. As input, the algorithm assumes a labeled DPN LN, the current trace σ, the newly observed activity a and its attribute values u^n, an SPN N^S, a control-flow alignment cost function κ, a variable cost function κ_{var}, a cache for OVAs Ω, the heuristic function h and the output of the previous execution of the algorithm for the process instance in question (i.e., the alignment function γ, the current variable values function α, the open set O, the closed set C, the cost-so-far function g and the predecessor function p). First, the states that have not been discovered yet are initialized (lines 3-5), then the outdated h values of the states in the open set are updated (i.e., the heuristic values are recalculated) thus the f values are updated too (lines 6-7). Next, the iterative search in the open set O starts (line 8). As the first step in the iteration, a state from the open set with the smallest f value is picked (line 9). If the state is a goal state (i.e., it contains a token in the last place of the trace net part) (line 10), then the optimal multi-perspective prefix-alignment $\bar{\gamma}$ was found, so it is returned along with the alignment function γ, the current variable assignment function α, the open set O, the closed set C, the cost-so-far function g, the predecessor function p and the cache for OVAs Ω (line 11). Otherwise, the current state is moved from the open- to the closed set (lines 12-13) and it is checked whether direct synchronizing (i.e., correct synchronous move) is possible (lines 14-34).

First, the variable values are checked and set as primes variables (lines 18-23). If all required variables are written and with valid values, then the algorithm moves forward to evaluate the guard function (lines 24-26). If the guard is satisfied (i.e., the guard function gives back a logical true value), then direct synchronizing is possible. If direct synchronizing is possible, then it can be executed (lines 27-34) and the searching process can stop (because the new optimal multi-perspective prefix-alignment $\bar{\gamma}$ was found). First, the new state m' is added to the open set O (line 28). Next, the optimal multi-perspective prefix-alignment γ of the previous state m is extended with the correct synchronous move, thus obtaining an optimal

multi-perspective prefix-alignment for the successor state m' (line 29). The values of the cost-so-far function g and the predecessor function p of the state m' are also set.

Algorithm 2: $A^{\star}_{inc}$ (Modified Incremental $A^{\star}$ Algorithm)

input : $LN = ((P, T, F, V, V_{dom}, V_{in}, T_{wr}, T_{gd}), \lambda_{act}, \lambda_{var}), \sigma \in (\mathcal{A} \times \mathcal{U}^n)^*, a \in \mathcal{A}$,
$u^n \in \mathcal{U}^n, N^S = (P^S, T^S, F^S, M_i^S, M_f^S, \lambda_{act}^S), O,C \subseteq \mathcal{R}(N^S, M_i^S)$,
$g : \mathcal{R}(N^S, M_i^S) \to \mathbb{R}_{\geq 0}, p : \mathcal{R}(N^S, M_i^S) \to T^S \times \mathcal{R}(N^S, M_i^S)$,
$h : \mathcal{R}(N^S, M_i^S) \to \mathbb{R}_{\geq 0}, \alpha : \mathcal{R}(N^S, M_i^S) \to \mathcal{U}^P_{V_{dom}}, \gamma : \mathcal{R}(N^S, M_i^S) \to \Gamma^*$,
$\kappa : \Gamma_c \to \mathbb{R}_{\geq 0}, \kappa_{var} : V \times \mathcal{U} \times V_{dom} \to \mathbb{R}_{\geq 0}$,
$\Omega : T^\star \times (\mathbb{N}^+ \times \mathcal{U}^P_{V_{dom}}) \to (\mathbb{N}^+ \times \mathcal{U}^P_{V_{dom}})$

1 **begin**
2 let $p_{|\sigma|}$ be the last place of the trace net part of N^S;
3 **forall** $m \in \mathcal{R}(N^S, M_i^S) \setminus O \cup C$ **do** // initialize undiscovered states
4 $g(m) \leftarrow \infty$;
5 $f(m) \leftarrow \infty$;
6 **forall** $m \in O$ **do**
7 $f(m) = g(m) + h(m)$; // recalculate heuristic and update f-values
8 **while** $O \neq \emptyset$ **do**
9 $m \leftarrow \arg\min_{m \in O} f(m)$; // pop a state with minimal f-value from O
10 **if** $p_{|\sigma|} \in_+ m$ **then**
11 **return** $\gamma(m), \gamma, \alpha, O, C, g, p, \Omega$;
12 $C \leftarrow C \cup \{m\}$;
13 $O \leftarrow O \setminus \{m\}$;
14 $dsp = False$; // check whether direct synchronizing is possible
15 **if** $\exists (t_t, t_m) \in T^S$ *s.t.* $(N^S, m)[(t_t, t_m)\rangle (N^S, m') \wedge \lambda^S_{act}((t_t, t_m)) = (a, a)$ **then**
16 $dsp = True$;
17 $V' \leftarrow \{\}$;
18 **forall** $v \in T_{wr}(t_m)$ **do** // check and set the prime variables
19 **if** $\pi_{\lambda_{var}(v)}(u^n) \in V_{dom}(v)$ **then**
20 $w(v') \leftarrow \pi_{\lambda_{var}(v)}(u^n)$;
21 $V' \leftarrow V' \cup v'$;
22 **else**
23 $dsp = False$;
24 **if** $dsp = True$ **then** // check guard function
25 $\alpha' = \alpha(m) \cup w$;
26 $dsp = eval_{V \cup V', V_{dom}}(T_{gd}(t_m), \alpha')$;
27 **if** $dsp = True$ **then** // do direct synchronizing if possible
28 $O \leftarrow O \cup \{m'\}$; // add the new state m'
29 $\gamma(m') \leftarrow \gamma(m) \cdot \langle((a, u^n), (t_m, w))\rangle$; // extend γ with the new move for m'
30 $g(m') \leftarrow g(m)$; // set cost to reach m'
31 $p(m') \leftarrow ((t_t, t_m), m)$; // set predecessor of m'
32 **forall** $v' \in dom(w)$ **do**
33 $\alpha(m')(v) \leftarrow w(v')$; // set α of m'
34 **return** $\gamma(m'), \gamma, \alpha, O, C, g, p, \Omega$;
35 **forall** $(t_t, t_m) \in T^S$ *s.t.* $(N^S, m)[(t_t, t_m)\rangle (N^S, m')$ **do**
36 **if** $m' \notin C$ **then** // investigate suc. states of m
37 $\gamma_c \leftarrow f_c(\gamma(m)) \cdot \langle(a, t_m)\rangle$; // extend cf. ver. of γ with the new move
38 $\gamma', g', \alpha', X \leftarrow obtMPPA(LN, \sigma, \gamma_c, \kappa, \kappa_{var}, \Omega)$; // obtain γ with OVAs
39 **if** $\gamma' \neq \langle\rangle$ **then** // a γ exists for γ_c
40 **if** $\exists n \in O$ *s.t.* $n = m' \wedge g(n) > g'$ **then** // a cheaper path to m' was found
41 $O \leftarrow O \setminus \{n\}$; // remove the old state n
42 **if** $m' \notin O$ **then**
43 $O \leftarrow O \cup \{m'\}$; // add the new state m'
44 $\gamma(m') \leftarrow \gamma'$; // add γ of m'
45 $g(m') \leftarrow g'$; // add cost to reach m'
46 $f(m') \leftarrow g(m') + h(m')$; // add f-value of m'
47 $p(m') \leftarrow ((t_t, t_m), m)$; // add predecessor of m'
48 $\alpha(m') \leftarrow \alpha'$; // add α of m'

Figure 3
Algorithm of the modified incremental A* algorithm

The value of the cost-so-far function g for state m' is the sum of the g value of the previous state m and the cost of the new alignment move. Since the new move is a correct synchronous move with zero cost, the value of g remains the same as in the previous state (line 30). The value of the predecessor function p for state m' is a tuple of the executed transition of the SPN (i.e., the synchronous move) and the previous state m (line 31). The variable assignments α are updated as well; the written variables get the values of the prime variables (lines 32-33). Finally, the expected values are returned (line 34).

If direct synchronizing is not possible, the search process continues by investigating the successor states of the current state m (only the ones that are not an element of the closed set C) (lines 35-47). For each successor state m', the respective control-flow alignment γ_c is obtained by extending the control-flow version of the multi-perspective prefix-alignment with OVAs γ of the previous state m with the alignment move that represents the SPN transition that leads from state m to m' (line 37). Next, the multi-perspective version with OVAs of the prefix-alignment is calculated (line 38) with the $obtMPPA$ function. The function expects a labeled DPN LN, a trace σ, a control-flow alignment γ_c, a control-flow cost function κ, a variable cost function κ_{var} and the cache for OVAs Ω as input. If an OVA exists for γ_c, it returns a multi-perspective prefix-alignment γ' (the control-flow prefix-alignment γ_c augmented with OVAs), the cost of the alignment g', the current variable values α' for state m' and the updated cache for OVAs Ω. If no OVA exists for γ_c, it returns an empty sequence. A state is added to the open set O only if there exists at least one valid multi-perspective prefix-alignment for it (i.e., γ' is not an empty sequence) (line 39), since the successor states of a state with no valid multi-perspective alignments have no valid multi-perspective alignments either. If the alignment is valid and there is a more expensive path for the same state m' in the open set O, then the more expensive path is removed (lines 40-41). To be more exact, the state is removed from the open set, then added again, thus the two cases (1) the state was not investigated before and (2) the state was investigated before, but a cheaper path was found can be handled with one code. Therefore, if state m' is not an element of the open set O, then it is added and the values of alignment function γ, the cost-so-far function g, the function f, the predecessor function p and the current variable values function α are set for state m' (lines 42-48).

3.4 Obtaining a Multi-Perspective Prefix-Alignment

In this section, the algorithm to obtain a multi-perspective prefix-alignment with OVAs is presented. There can be multiple OVAs for a control-flow prefix-alignment and the algorithm chooses only one for it. In case there is no OVAs for the given prefix-alignment, then an empty sequence is returned instead to indicate this problem.

In *Figure 4* the algorithmic description of obtaining a multi-perspective prefix-alignment with OVAs is presented. As input, the algorithm assumes a labeled DPN LN, the current trace σ, a control-flow prefix-alignment γ_c, a control-flow cost

function κ, a variable cost function κ_{var} and a cache for OVAs Ω. As output, it returns a multi-perspective prefix-alignment with OVAs γ, the cost-so-far g, the current variable assignment α, and the (updated) cache for OVAs Ω.

In the first main step of the algorithm, the identifiers of the OVA problem, the control-flow model steps ρ (i.e., the sequence of the fired transitions of LN according to the control-flow alignment γ_c) and the variable writings w_σ (according to the current trace σ) are generated (lines 2-16). The reason why ρ and w_σ are used as the OVA problem identifiers is that every OVA problem is uniquely defined by the guard functions and the variable values written by the fired transitions. Since every transition can have at most one guard function, it is enough to cache only the fired transitions and the observed written values for each variable at each transition firing. The process of obtaining ρ and w_σ is done by iterating in the control-flow alignment γ_c with the indexes i and j. These indexes identify the sequence number of the log step and process step within the currently examined alignment move, respectively. First, the necessary objects are initialized: the control-flow model steps ρ (line 2), the variable writings w_σ (line 3) and the indexes i and j (lines 4-5). Next, the iteration starts in γ_c (line 6). If the current alignment move involves a log step (line 7), then index i is increased by one, thus obtaining the index of the current log step (line 8). Likewise, if the current alignment move involves a process step (line 9), then the index j is increased by one, thus obtaining the index of the current process step (line 10). Furthermore, the sequence of the control-flow model steps ρ is extended with transition t of the current alignment move (line 11), then the algorithm checks whether the transition has variable writing operations assigned to it and whether the current alignment move is synchronous (line 12). If the transition has variable writing operations and is within a synchronous alignment move, then for each written variable v the written value u is extracted from the current trace σ (lines 13-14). If the written value u is in the domain of the variable v (line 15), then in the j-th process step the value u is assigned to variable v within w_σ (line 16).

In the second main step of the algorithm, the solution for the OVA problem is obtained (lines 17-23). If the OVA problem was solved before (i.e., the OVA problem identifiers are in the domain of the cache for OVAs Ω) (line 18), then the solution ω (i.e., the OVAs) is loaded from the OVA cache Ω by using the problem identifiers that were generated in the previous steps, the control-flow model steps ρ and the variable writings w_σ (line 19). If the OVA problem was not solved before, then from the given input, the OVA problem is created as a MILP problem with the $createOVAP$ function (line 21). This function returns the components of the MILP problem: the variables VAR, the constraints $CSTR$ and the objective function OBJ. Afterward, it is solved with the $solveOVAP$ function (line 22). Within this function, the algorithm calls an arbitrary MILP solver to solve the problem, and then returns the variable assignments of the optimal solution (i.e., a function ω that assigns the optimal value to each variable in VAR at each process step). The obtained result ω is added to the cache of OVAs Ω as a solution to the OVA problem that is defined by ρ and w_σ (line 23). If the problem has no optimal solution, then ω stays an empty

set (line 24) and the algorithm returns an empty sequence as γ to indicate that no OVA exists for the control-flow alignment γ_c. Furthermore, it returns 0 as g, an empty set as α, and the unchanged cache of OVAs Ω, just to have the same number of outputs as in the other cases (line 25).

Algorithm 3: $obtMPPA$ (Obtain Multi-Perspective Prefix-Alignment with OVAs)

input : $LN = ((P, T, F, V, V_{dom}, V_{in}, T_{wr}, T_{gd}), \lambda_{act}, \lambda_{var}), \sigma \in (\mathcal{A} \times \mathcal{U}^n)^*$,
$\gamma_c : \mathcal{R}(N^S, M_i^S) \to \Gamma_c^*, \kappa : \Gamma_c \to \mathbb{R}_{\geq 0}, \kappa_{var} : V \times \mathcal{U} \times V_{dom} \to \mathbb{R}_{\geq 0}$,
$\Omega : T^\star \times (\mathbb{N}^+ \times \mathcal{U}_{V_{dom}}^P) \to (\mathbb{N}^+ \times \mathcal{U}_{V_{dom}}^P)$

1 **begin**
2 $\rho \leftarrow \langle\rangle$; // initialize cache for model steps (sequence of transitions)
3 $w_\sigma \leftarrow \emptyset$; // initialize cache for variable writings (from σ)
4 $i \leftarrow 0$; // initialize index for log steps in γ_c
5 $j \leftarrow 0$; // initialize index for process steps in γ_c
6 **forall** $(a, t) \in \gamma_c$ **do** // collect the variable writings for each process step
7 **if** $a \neq \gg$ **then**
8 $i \leftarrow i + 1$;
9 **if** $t \neq \gg$ **then**
10 $j \leftarrow j + 1$;
11 $\rho \leftarrow \rho \cdot \langle t \rangle$;
12 **if** $T_{wr}(t) \neq \emptyset \wedge a = \lambda_{act}(t)$ **then**
13 **forall** $v \in T_{wr}(t)$ **do**
14 $u \leftarrow \pi_{\lambda_{var}(v)}(\pi_2(\pi_i(\sigma)))$;
15 **if** $u \in V_{dom}(v)$ **then**
16 $w_\sigma(v_j) \leftarrow u$;
17 $\omega \leftarrow \emptyset$;
18 **if** $(\rho, w_\sigma) \in dom(\Omega)$ **then** // if the OVA problem was solved before
19 $\omega \leftarrow \Omega(\rho, w_\sigma)$; // load the OVA problem solution
20 **else** // if the OVA problem was NOT solved before
21 $VAR, CSTR, OBJ \leftarrow createOVAP(LN, \rho, w_\sigma)$; // create the OVA problem
22 $\omega \leftarrow solveOVAP(VAR, CSTR, OBJ)$; // solve the OVA problem
23 $\Omega(\gamma_c, w_\sigma) \leftarrow \omega$; // save the OVA problem solution
24 **if** $\omega = \emptyset$ **then**
25 **return** $\langle\rangle, 0, \emptyset, \Omega$
26 $\gamma \leftarrow \langle\rangle$; // initialize the multi-perspective prefix-alignment γ
27 $g_c, g_{var}, i, j \leftarrow 0$; // initialize the cost-so-far g_c, g_{var} and the indexes i and j
28 **forall** $v \in V$ **do** // add all the variables with initial values to α
29 $\alpha(v) \leftarrow V_{in}(v)$;
30 **forall** $(a, t) \in \gamma_c$ **do** // create γ from γ_c
31 $s_l, s_m \leftarrow \gg$;
32 **if** $a \neq \gg$ **then** // if the move involves a log step
33 $i \leftarrow i + 1$;
34 $u^n \leftarrow \pi_2(\pi_i(\sigma))$;
35 $s_l \leftarrow (a, u^n)$; // augment log step with attribute values
36 **if** $t \neq \gg$ **then** // if the move involves a process step
37 $j \leftarrow j + 1$;
38 $w \leftarrow \emptyset$;
39 **forall** $v \in T_{wr}(t)$ **do**
40 $w(v) \leftarrow \omega(v_j)$; // get OVA for variable v
41 $\alpha(v) \leftarrow w(v)$; // update the value of variable v within α
42 $g_{var} \leftarrow g_{var} + \kappa_{var}(v, w_\sigma(v_j), w(v))$; // add the cost of the variable to g_{var}
43 $s_m \leftarrow (t, w)$; // augment process step with variable assignments
44 $g_c \leftarrow g_c + \kappa(a, \lambda_{act}(t))$; // add the cost of the (control-flow) alignment step to g_c
45 $\gamma \leftarrow \gamma \cdot \langle (s_l, s_m) \rangle$; // extend γ with the mp alignment move
46 $g \leftarrow g_c + g_{var}$; // get the cost of the multi-perspective prefix-alignment γ
47 **return** $\gamma, g, \alpha, \Omega$

Figure 4

Algorithm of obtaining a Multi-Perspective Prefix-Alignment with OVAs

In the third main step of the algorithm, a multi-perspective prefix-alignment γ is generated from the control-flow prefix-alignment γ_c while the cost-so-far g and the current variable assignments α are also calculated (lines 26-46). First, the

components of the procedure are initialized. The multi-perspective prefix-alignment γ is initialized as an empty sequence (line 26). The cost-so-far of the control-flow alignments g_c, the cost-so-far of the variables assignments g_{var}, and the indexes i and j (to iterate in the control-flow prefix-alignment γ_c and the current trace σ, respectively) are initialized as 0 (line 27). The current variable assignments α is initialized by assigning to each variable their initial value (lines 28-29). Next, the iteration starts in the control-flow alignment γ_c (line 30) to augment each alignment move with the attribute values (if it involves a log step) (lines 32-35) and the OVAs (if it involves a process step) (lines 36-43), to obtain a multi-perspective alignment move. If the alignment move involves a process step with variable writings, the new values (the OVAs for that alignment move) overwrite the previous values in α (line 41), thus at the end of the iteration, the current variable assignments are obtained. In addition, to calculate g later, the cost of the variable assignment is added to g_{var} (line 42). Furthermore, in each iteration step, the cost of the control-flow alignment move is added to g_c (line 44), thus at the end of the iteration, the cost-so-far of multi-perspective prefix-alignment g is obtained as the sum of g_{var} and g_c (line 46). Similarly, in each iteration step, γ is extended with the obtained multi-perspective alignment move (line 45), thus at the end of the iteration, the multi-perspective prefix-alignment is obtained. After the iteration finishes, the necessary objects (γ, g, and α) are obtained, hence they are returned along with the updated Ω (line 47).

In the presented algorithmic description only the ω is cached to minimize the length of the description, but g_{var} and α could be stored in the cache of OVAs Ω as well. This is the reason why the value of g is stored separately in g_c and g_{var}. If g_{var} and α are cached too and the OVA problem was solved before, then only the multi-perspective prefix-alignment γ needs to be generated. Consequently, the algorithm of it is very similar to the third main step of the presented algorithm, except that α and g_{var} are not calculated (since they are obtained from the cache).

The algorithmic description of creating an OVA problem as a MILP problem is presented [9].

4 Real Dataset Evaluation

The proposed approach (hereinafter MOCC) was implemented in Python 3.7 as an extension to version 1.5.2.2 of PM4Py [26] (PM for Py) and tested in version 3.3.6 of Spyder development environment. PM4Py is a process mining package for Python, where most of the process mining algorithms are implemented. To solve the MILP problem for the OVA problem, Google OR-Tools [27] was used. The tests were performed on a laptop PC, with Windows 10 operation system, equipped with an Intel(R) Core(TM) i5-3320M, 2.60 GHz 2-core CPU and 8 GB of RAM.

The source code of the incremental A* algorithm [28] was modified, then got integrated into the above-mentioned version of PM4Py. Moreover, program codes were implemented, which allow importing DPN process model from PNML file and which can generate and solve a MILP problem to find the OVAs for the given variable writings and guard functions.

The implemented MOCC solution was tested on real-life datasets of three real processes: a manufacturing process, a road traffic fine management process [30] and a hospital billing process [31]. All the three datasets were processed and divided into two kinds of event logs: one with only good process executions (i.e., all the traces fit completely the process model) and one with only wrong process executions (i.e., neither of the traces fit completely the process model).

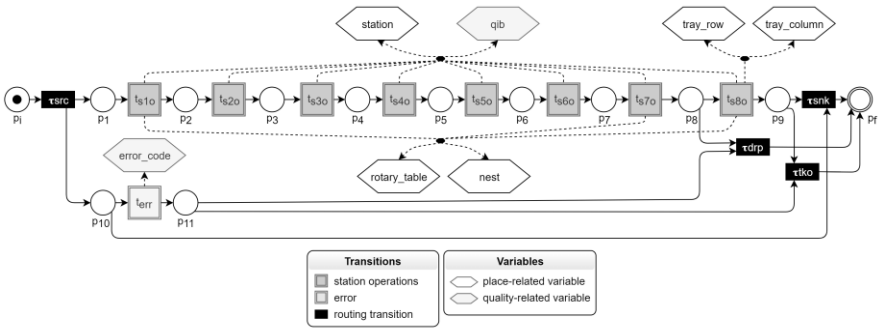
The MOCC solution and the BMCC solution [8] were applied to the fine management process and the billing process to compare the resulting alignments of the two methods with each other. Since the MOCC calculates multi-perspective prefix-alignments and the BMCC multi-perspective alignments, some differences are expected between the results. It is important, that both processes were examined before with the BMCC solution in [9], because this way it is known what results should be obtained for the given inputs by applying the method. The BMCC solution was applied as the "Conformance Checking of DPN" plugin from the "DataAwareReplayer" package in ProM 6.9 [29].

The MOCC solution and the OCC solution [18] were applied on the manufacturing process, to measure and then compare their computation time with each other for the same input. In contrast to the other two processes, the manufacturing process has a short lead time (few minutes at most), thus it is very important how fast the output of the MOCC solution can be obtained (i.e., how fast the deviations from the process model can be detected).

4.1 The Process Models

The DPN process models were created with the help of "Create DPN (Text language based)" plugin in ProM 6.9 then exported into a PNML files. Only one small change was made in the output files: an initial value was added to each variable.

The DPN process models of the fine management process and the billing process were created based on the models that are presented in [9]. For this reason, they are not included in this paper. The created DPN process model of the manufacturing process is shown on *Figure 5*. It can be seen that it is quite complex process, since it has 7 process variables, 25 variable writing operations and 12 complex guard functions. For a detailed description of the process, see our previous work [32].



Transition	Guard expression
t_{s1o}	$\text{rotary\_table}' = \text{"M"} \wedge \text{nest}' \geq 1 \wedge \text{nest}' \leq 8$
$t_{s2o}, t_{s3o}, \ldots, t_{s6o}$	$(\text{qib} = 1 \wedge \text{error\_code} = 0) \vee (\text{qib} = 0 \wedge \text{qib}' = 0 \wedge \text{error\_code} \neq 0)$
t_{s7o}	$\big((\text{qib} = 1 \wedge \text{rotary\_table}' = \text{"S"} \wedge \text{nest}' \geq 1 \wedge \text{nest}' \leq 4) \vee (\text{qib} = 0 \wedge \text{rotary\_table} = \text{"M"} \wedge \text{nest}' \geq 1 \wedge \text{nest}' \leq 8)\big) \wedge$ $\big((\text{qib} = 1 \wedge \text{error\_code} = 0) \vee (\text{qib} = 0 \wedge \text{qib}' = 0 \wedge \text{error\_code} \neq 0)\big)$
t_{s8o}	$(\text{rotary\_table}' = \text{"M"} \wedge \text{nest}' \geq 1 \wedge \text{nest}' \leq 8) \wedge$ $(\text{tray\_row}' > 0 \wedge \text{tray\_column}' > 0) \wedge$ $\big((\text{qib} = 1 \wedge \text{error\_code} = 0) \vee (\text{qib} = 0 \wedge \text{qib}' = 0 \wedge \text{error\_code} \neq 0)\big)$
t_{err}	$\text{error\_code}' \geq \text{station} * 100 + 1 \wedge$ $\text{error\_code}' < \text{station} * 100 + 100 \wedge \text{qib} = 0$
t_{drp}	$\text{station} = 7 \wedge \text{qib} = 0 \wedge \text{error\_code} < 700$
t_{tko}	$\text{station} = 8 \wedge \text{qib} = 0 \wedge \text{error\_code} > 700$
t_{snk}	$\text{station} = 8 \wedge \text{qib} = 1 \wedge \text{error\_code} = 0$

Figure 5
DPN process model of the examined process

4.2 Experiment Setup

The implemented solution was tested on the following kind of event logs for each process:

1) One, which only contains traces that perfectly fit the model (hereinafter good traces). This means the fitness values for all traces are exactly 1 and the total cost of their optimal multi-perspective alignments are 0.

2) One, which only contains traces that do not fit the model (hereinafter bad traces). This means the fitness values for all traces are less than 1 and the total cost of their optimal multi-perspective alignments are greater than 0.

The number of cases and events varies within the event logs. For the manufacturing process, for every trace in both files, multiple cases were included. (Different cases can have the same trace.) This way 67 cases with fitting traces and 85 cases with unfitting traces were collected. For the other two processes, since they have too many kinds of traces, only the 10 most common traces were taken into account and 5 cases for each were collected in both files (i.e., 50-50 cases in total).

The OCC solution was applied on the control-flow copy of the DPN process model (i.e., simple Petri net) of the manufacturing process. The computation time of both solutions was compared with each other. It is expected that the MOCC solution is slower, but still relatively fast (e.g., less than 1 second per case).

The MOCC solution is expected to give the same results as the BMCC solution for the same input, except for the incomplete traces. In an online setting incomplete traces are not seen as wrong behaviors, because the process still can be ongoing.

4.3 Result Analysis

Both the OCC and the MOCC methods were executed 10 times for both event logs of the manufacturing process.

Table 1
Statistics of the execution times of OCC and MOCC for the different event data (in seconds)

	Good traces		**Bad traces**	
	OCC	**MOCC**	**OCC**	**MOCC**
Sum (s)	17.0545	3.5478	21.6476	595.3002
Mean (s)	0.2545	0.0530	0.2547	7.0035
Std. deviation (s)	0.0353	0.0193	0.0977	6.9647
Minimum (s)	0.1999	0.0430	0.0220	0.0160
Maximum (s)	0.3178	0.1999	0.4087	29.5627

The statistics of the average execution times in seconds are summarized in Table 1 for both good traces and bad traces. Unexpectedly, for good traces, the MOCC solution was faster than the OCC solution. This is due to that MOCC allows direct synchronizing, thus fewer states are examined during the search process compared to the OCC. In contrast, for bad traces, the MOCC solution was much slower. This is due to that less direct synchronizing could be made and calculations for finding OVAs had to be executed. As it could be seen in Table 1 too, the execution time per case for bad traces at MOCC was quite varied. This is due to that OVA cache decreased the execution time for traces with repetitive OVA problem.

Based on the output of the MOCC solution, the most common wrong behaviors of the manufacturing process are related to how and when does a product get removed from the assembly line. In the case of a finished product, the written values to the *tray_row* and the *tray_column* variables should be greater than 0, representing that the product was placed on a tray. Otherwise, in the output prefix-alignment, the transition and event for Station 8 operations appear in form of an incorrect synchronous move, which can be interpreted as the product being discarded. In the case of an unfinished faulty product, the last recorded event should be for Station 7 operations, representing that the product was discarded at Station 7. If there are any following events recorded, they appear in form of log moves in the output prefix-alignment, which can be interpreted as the product was not discarded at Station 7.

However, if the last recorded event is for a previous station, the whole trace appears as an unfinished good execution in the output prefix-alignment, even if it should be interpreted as the product was discarded at a wrong station.

It must be noted that good behaviors could be wrongly recorded as wrong behaviors. For this reason, whenever the above-mentioned cases occur in real life, it should be checked by the machine operator whether the source of the problem is the tool that replaces the product, the sensor that provides information, or the software that records the data. It is also a possibility that the source of the problem is the occurrence of a not modeled event (e.g., the machine operator removed a product from the assembly line).

Both the MOCC and the BMCC methods were executed for the road traffic fine management process and the hospital billing process. As it was expected, it gave nearly the same results for the completed traces as the BMCC solution, but in an online environment. There were differences in whether a log step is followed by a process step or a process step is followed by a log step. However, this difference is insignificant, since it has no effect on the total cost of the alignment. Furthermore, in the case of MOCC, the invisible transitions at the end of the process model were not executed. This is due to that it thinks that the process instance is still ongoing.

In summary, the MOCC solution met the expectations for accuracy as a multi-perspective solution. On completed traces and process models without invisible transitions at the end, the same multi-perspective alignments can be obtained with it as with the BMCC solution, but faster. Furthermore, since the MOCC solution can process incomplete traces too, it can give information about the conformance of process instances that are still running. The MOCC solution (due to its multi-perspective view on the process) can detect most of the wrong behaviors, but (due to its online view on the process) it does not know when a process execution ends. However, this problem can easily be fixed by using a time limit. If no new events are observed for a case within the set time limit, the process instance is automatically declared as finished, or the system asks for confirmation. The former strategy could be used for cases where the last event is a transition that directly leads to a finished state in the process model, and the latter strategy could be used for the other cases.

Conclusions

In this paper, a prefix-alignment based MOCC solution, was developed to support the real-time monitoring of event data, from various angles. A modified version of the incremental A* algorithm was developed, that solves OVA problems, to find an optimal multi-perspective prefix-alignment for the running, unfinished cases. To decrease execution times, direct synchronizing and using a cache for OVA problems was added to the algorithm. The MOCC solution was implemented and tested with multiple real-life processes. The results show that it can meet the expectations of both online and multi-perspective (prefix-)alignment calculation methods.

Therefore, with the correct settings, it can be used to monitor real-life processes that have a short lead time and a prescriptive process model.

In the future, the solution will be improved upon, to further decrease its execution time, for traces that do not completely fit the model.

Acknowledgement

The research is part of project TKP2020-NKA-10 that has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme no. 2020-4.1.1.-TKP2020 (National Challenges Subprogramme) funding scheme. The authors acknowledge financial support also from the Slovenian–Hungarian bilateral project "Optimization and fault forecasting in port logistics processes using artificial intelligence, process mining and operations research", grant 2019-2.1.11-TÉT-2020-00113, and from the National Research, Development and Innovation Office–NKFIH under the grant SNN 129364.

References

[1] W. M. P. van der Aalst. Process mining: discovery, conformance and enhancement of business processes, Vol. 2, Heidelberg: Springer, 2011, doi:10.1007/978-3-642-19345-3

[2] J. Carmona, B. van Dongen, A. Solti, and M. Weidlich. Conformance checking. Cham: Springer, 2018, doi:10.1007/978-3-319-99414-7

[3] S. Dunzer, M. Stierle, M. Matzner, and S. Baier. Conformance checking: a state-of-the-art literature review. In: Proceedings of the 11th International Conference on Subject-Oriented Business Process Management, 2019, pp. 4:1-10, doi:10.1145/3329007.3329014

[4] N. Sidorova, C. Stahl, and N. Trčka. Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. In: Information Systems, 2011, 36(7), pp. 1026-1043, doi:10.1016/j.is.2011.04.004

[5] M. De Leoni, and W. M. P. van der Aalst. Data-aware process mining: discovering decisions in processes using alignments. In: Proceedings of the 28th annual ACM symposium on applied computing, 2013, pp. 1454-1461, doi:10.1145/2480362.2480633

[6] M. De Leoni, W. M. P. van Der Aalst, and B. F. van Dongen. Data-and resource-aware conformance checking of business processes. In: International Conference on Business Information Systems, Springer, Berlin, Heidelberg, 2012, pp. 48-59, doi:10.1007/978-3-642-30359-3_5

[7] M. De Leoni, and W. M. P. van der Aalst. Aligning event logs and process models for multi-perspective conformance checking: An approach based on integer linear programming. In: Business Process Management. Springer, Berlin, Heidelberg, 2013, pp. 113-129, doi:10.1007/978-3-642-40176-3_10

[8] F. Mannhardt, M. De Leoni, H. A. Reijers, and W. M. P. van der Aalst. Balanced multi-perspective checking of process conformance. In: Computing 98.4, 2016, pp. 407-437, doi: 10.1007/s00607-015-0441-1

[9] F. Mannhardt. Multi-perspective Process Mining. Eindhoven: Technische Universiteit Eindhoven, Doctoral Thesis, SIKS disertation series, Vol. 2018-02, 2018

[10] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst. Conformance checking using cost-based fitness analysis. In: 2011 IEEE 15th International Enterprise Distributed Object Computing Conference IEEE, 2011, pp. 55-64, doi:10.1109/EDOC.2011.12

[11] A. Adriansyah. Aligning observed and modeled behavior. Doctor of Philosophy, Department of Mathematics and Computer Science, Eindhoven, 2014, doi:10.6100/IR770080

[12] S. J. van Zelst, A. Bolt, and B. F. van Dongen. Tuning Alignment Computation: An Experimental Evaluation. In: Proceedings of ATAED 2017, 2017, pp. 1-15

[13] S. J. van Zelst, A. Bolt, and B. F. van Dongen. Computing alignments of event data and process models. In: Transactions on Petri Nets and Other Models of Concurrency XIII. Springer, Berlin, Heidelberg, 2018, pp. 1-26, doi: 10.1007/978-3-662-58381-4_1

[14] F. Taymouri. Light methods for conformance checking of business processes. Tesi doctoral, UPC, Departament de Ciències de la Computació, 2018

[15] A. Burattin, and J. Carmona. A framework for online conformance checking. In: International Conference on Business Process Management. Springer, Cham, 2017, pp. 165-177, doi:10.1007/978-3-319-74030-0_12

[16] A. Burattin, S. J. van Zelst, A. Armas-Cervantes, B. F. van Dongen, and J. Carmona. Online conformance checking using behavioural patterns. International Conference on Business Process Management. Springer, Cham, 2018, pp. 250-267, doi:10.1007/978-3-319-98648-7_15

[17] S. J. van Zelst, A. Bolt, M. Hassani, B. F. van Dongen, and W. M. P. van der Aalst. Online conformance checking: relating event streams to process models using prefix-alignments. In: International Journal of Data Science and Analytics 8.3, 2019, pp. 269-284, doi:10.1007/s41060-017-0078-6

[18] D. Schuster, and S. J. van Zelst. Online Process Monitoring Using Incremental State-Space Expansion: An Exact Algorithm. arXiv preprint arXiv:2002.05945, 2020, doi: 10.1007/978-3-030-58666-9_9

[19] W. L. J. Lee, A. Burattin, J. Munoz-Gama, and M. Sepúlveda, Orientation and conformance: A HMM-based approach to online conformance checking. In: Information Systems, 101674, 2020, doi:10.1016/j.is.2020.101674

[20] D. Schuster, and G. J. Kolhof. Scalable Online Conformance Checking Using Incremental Prefix-Alignment Computation. arXiv preprint arXiv:2101.00958, 2020

[21] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In: IEEE transactions on Systems Science and Cybernetics, 4(2), 1968, pp. 100-107, doi: 10.1109/TSSC.1968.300136

[22] V. Bloemen, J. van de Pol, and W. M. P. van der Aalst. Symbolically aligning observed and modelled behaviour. In: 2018 18th International Conference on Application of Concurrency to System Design (ACSD), IEEE, 2018, pp. 50-59, doi:10.1109/ACSD.2018.00008

[23] A. Schrijver. Theory of linear and integer programming. John Wiley & Sons, New York, 1998

[24] S. P. Bradley, A. C. Hax, and T. L. Magnanti. Applied Mathematical Programming. Addison-Wesley, 1977

[25] Z. Nagy and Á. Werner-Stark. A Multi-perspective Online Conformance Checking Technique. In: 2020 6th International Conference on Information Management (ICIM), IEEE, 2020, pp. 172-176, doi:10.1109/ICIM49319.2020.244693

[26] S. van Zelst. PM4Py – Process Mining for Python. Date of last access: 2020.11.30. https://pm4py.fit.fraunhofer.de/

[27] Google Inc. OR-Tools - Google Optimization Tools. Date of last access: 2020.11.30. https://developers.google.com/optimization/

[28] D. Schuster, and S. van Zelst. GitHub - fit-daniel-schuster / online_process_monitoring_using_incremental_state-space_expansion_an_exact_algorithm. Date of last access: 2020.11.30. https://github.com/fit-daniel-schuster/online_process_monitoring_using_incremental_state-space_expansion_an_exact_algorithm

[29] ProM Tools. ProM 6.9. Date of last access: 2021.06.15. http://www.promtools.org/doku.php?id=prom69

[30] M. de Leoni, and F. Mannhardt. Road Traffic Fine Management Process. 4TU.ResearchData, Dataset, 2015, https://doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5

[31] F. Mannhardt. Hospital Billing - Event Log. Eindhoven University of Technology, Dataset, 2017, https://doi.org/10.4121/uuid:76c46b83-c930-4798-a1c9-4be94dfeb741

[32] Z. Nagy, A. Werner-Stark, and T. Dulai. Using Process Mining in Real-Time to Reduce the Number of Faulty Products. In: European Conference on Advances in Databases and Information Systems. Springer, Cham, 2019, pp. 89-104, doi: 10.1007/978-3-030-28730-6_6