Optimising Event Pattern Matching using Business Process Models

Matthias Weidlich, *Member, IEEE,* Holger Ziekow, Avigdor Gal, *Senior Member, IEEE,* Jan Mendling, Mathias Weske, *Member, IEEE*

Abstract—A growing number of enterprises use complex event processing for monitoring and controlling their operations, while business process models are used to document working procedures. In this work, we propose a comprehensive method for complex event processing optimisation using business process models. Our proposed method is based on the extraction of behavioural constraints that are used, in turn, to rewrite patterns for event detection, and select and transform execution plans. We offer a set of rewriting rules that is shown to be complete with respect to the all, seq, and any patterns. The effectiveness of our method is demonstrated in an experimental evaluation with a large number of processes from an insurance company. We illustrate that the proposed optimisation leads to significant savings in query processing. By integrating the optimisation in state-of-the-art systems for event pattern matching, we demonstrate that these savings materialise in different technical infrastructures and can be combined with existing optimisation techniques.

Index Terms—Event processing, query optimisation, query rewriting

1 Introduction

With the increase in data and monitoring facilities availability, event processing becomes more central in enterprises. Business activity monitoring (BAM) [1] and management of service level agreements (SLA) [2] build on event processing technology to aggregate events and detect complex event patterns over event streams created by business process executions.

Event processing of business process data poses high agility and scalability requirements due mainly to the number of process steps (often more than 20), the concurrent execution of sometimes hundreds of instances, and the multiple assertions among events that characterize processing states. The scalability challenge was highlighted in previous studies [3], [4], indicating that there is a major performance degradation as application's complexity increases, thereby evidencing that optimisation efforts comprise a significant value.

The need for algorithmic optimisation of event processing in a business process setting becomes acute due to the lack of sufficient human expertise. Business analysts conducting BAM or managing SLAs rarely have expert knowledge on the actual process implementation and find it hard to harness implementation details to the benefit of event processing. A survey conducted by *ebizQ* revealed that in 84% of the investigated cases, not IT experts, but "business analysts and business specialists, such as quantitative analysts, compliance analysts and risk managers, will define event-processing rules" [5]. Thus, whenever event patterns are defined by analysts with limited insights on how a process is implemented, inefficiencies are likely. Also, business experts prefer simple, high-level descriptions, e.g., templates for structured text as reported by Magid et al. [6], which are translated into event query patterns. The latter, just like the use of views in databases, may introduce inefficiencies in the way event patterns are defined.

This research was partially funded by FP7/2007-2013 under grant agreement 318275 (GET Service).

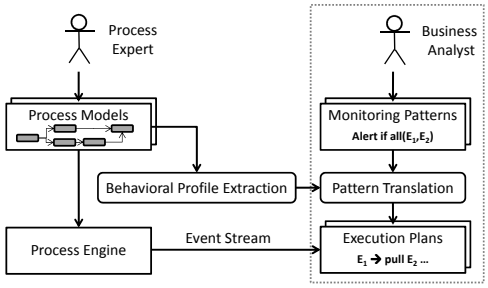

Fig. 1: Process-tailored event processing optimisation.

In this work, we propose a method for optimising event processing using knowledge about potential event orderings. It targets situations where knowledge about business processes is readily available in the form of process models, serving as a basis for process automation [7]. Given the discussion above, in such a setting one cannot assume that event patterns are defined in the most efficient way given a particular implementation of a process. Instead, process knowledge shall be used for tailoring event processing, as illustrated in Fig. 1. The left part of that figure depicts how a business process is enacted by a process expert with a technical workflow model. On the right, a business analyst defines patterns to detect particular complex events. These patterns are translated into execution plans.

To illustrate our approach, consider, for example, a shipment process from the logistics domain. A process engine that executes that process may first book a route in parallel to confirming shipment, which is followed by either scheduling a regular or a fast lane delivery. Position tracking is then executed, possibly repeated multiple times, towards completion of the process. For this setting, a BAM task may investigate whether a pre-booked route is followed, which translates into a conjunction event pattern. Such a pattern refers to the joint occurrence of route booking and position tracking, i.e., a complex event materializes as soon as two events of either type

are observed. Based on the information from the process model, however, we may deduce that events of these types are always sequentially ordered. Consequently, the conjunction pattern can be rewritten to a sequence pattern, matching a particular order of events of both types, to enable more efficient processing. An SLA, in turn, may relate to the time elapsed between the shipment confirmation and the initialization of fast lane delivery, leading to an event sequence pattern. Executing such a sequence pattern may be done push-based (wait for the delivery event once a confirmation event occurred) or pull-based (try to fetch a confirmation event upon the occurrence of a delivery event). Knowing from the process model that confirmation is only sometimes succeeded by fast line delivery, but fast lane delivery is always preceded by confirmation, selecting of a pull-based execution plan helps to achieve efficient processing.

To leverage the knowledge encoded in a process model, our approach builds on the formal concept of a behavioural profile [8], [9] that covers constraints about activity execution ordering, mutual exclusion, and co-occurrence of activities. For event patterns that relate to execution of business activities, we use these constraints for optimisation at three stages. First, patterns are rewritten without changing their semantics in terms of the result set. Second, constraints guide the selection of execution plans to avoid the anticipation of invalid event sequences. Third, execution plans are rewritten to optimise performance. We define the requirements of optimisation rules and present a set of such rules that is complete for the conjunction (all), sequence (seq), and disjunction (any) operators under the behavioural profile model.

This paper makes the following contributions.

- We introduce a generic multi-step approach to pattern optimisation that advances the state-of-the-art by utilizing process knowledge on the event sources.
- A set of optimisation rules is presented and formalized. We prove correctness, completeness, and efficiency of the presented rules, and discuss their joint application in detail.
- The paper reports on a comprehensive evaluation of the approach. We study applicability and explore potential benefits with a large number of processes from an insurance company and queries that relate to SLAs for these processes.
- We demonstrate memory savings obtained using the proposed method for realistic workloads in two sample applications based on Streams,[1] a stream processing framework, and Esper,[2] a publicly available event processing engine. We also highlight the orthogonality of our method to common optimisation techniques and combine our approach with optimisations proposed by Wu et al. [18].

We sketched the idea for process-based optimisation in [10], providing a partial set of rules that is applicable only for conjunctive patterns without the any operator, and lacks proofs of correctness and efficiency for the rules. Also, we offer in this work an empirical evaluation that was not done before.

The remainder of the paper is structured as follows. Section 2 provides background on process models. Section 3 introduces basic terminology and concepts of complex event processing and presents event orderings, tying together process knowledge and event types. Section 4 introduces our approach to optimisation, grounded in behavioural profiles of process models. Section 5 presents the evaluation of this approach. Section 6 discusses related work, before Section 7 concludes the paper.

1. http://www.jwall.org/streams/
2. http://esper.codehaus.org/

2 Background: Process Models

Process models are extensively used in companies for describing business operations and technical process implementations. Such a model directs the flow of execution for instances of a business process. It may be used by a process engine as a template for execution [11], thereby playing a normative role. Figure 2 depicts a process model of the logistics process described in the introduction, specified using BPMN [12].

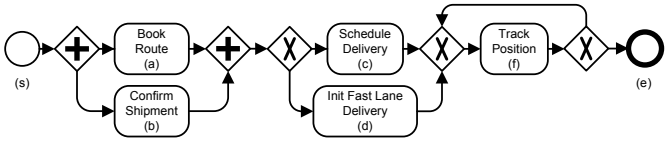

Fig. 2: Process model defined in BPMN.

Formally, a process model can be seen as a graph containing activity nodes and control nodes.

Definition 1 (Process Model): A *process model* is a sextuple $P = (A, s, e, C, F, T)$ where:

- A is a finite non-empty set of activity nodes (activities),
- C is a finite set of control nodes,
- $N = A \cup C$ is a finite set of nodes with $A \cap C = \emptyset$,
- $F \subseteq N \times N$ is a flow relation, such that (N, F) is a connected graph,
- $\bullet n = \{n' \in N \mid (n', n) \in F\}$, $n\bullet = \{n' \in N \mid (n, n') \in F\}$ denote predecessors and successors. We require $\forall\ a \in A : |\bullet a| \leq 1 \wedge |a \bullet| \leq 1$,
- $s \in A$ is the only start node, such that $\bullet s = \emptyset$,
- $e \in A$ is the only end node, such that $e\bullet = \emptyset$,
- $T : C \rightarrow \{and, xor\}$ assigns types to control nodes.

Definition 1 separates activity nodes (A) from control nodes (C). Activity nodes are constrained to have at most one successor and one predecessor while control nodes can have multiple successors and predecessors. The start and end activity nodes are dedicated nodes that indicate initialization and termination of a process. For a process model, we assume trace semantics that are defined by a translation into a Petri net following common formalizations, cf., [13]. With such a translation, the behaviour of a process model $P = (A, s, e, C, F, T)$ is represented as a set of completed *traces* $\mathcal{T}_P$. This set includes sequences of the form $s \cdot A^* \cdot e$ with A^* being the set of all finite sequences over A. Those sequences represent the execution order of activities. For the example given in Fig. 2, for instance, the set of completed traces includes the traces (s, a, b, c, f, f, f, e) and (s, b, a, d, f, e).

A *process instance* is a single execution of a business process, e.g., a shipment process executed for a specific origin, target, and product, and is represented by a completed trace.

3 MODEL

This section presents a CEP model that is geared towards events that stem from process models. Our terminology is based on the event processing language as defined by Etzion and Niblett [14]. Events are discussed in Section 3.1, followed by a discussion on the syntax and semantics of patterns (Section 3.2). Section 3.3 introduces execution plans. Finally, we tie the model to process behavioural profiles in Section 3.4.

3.1 Events

An *event* e is an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain, cf., [14]. As such, we assume point-based semantics of events. An event type $E \in \mathbb{E}$ is a specification for a set of events that share the same semantic intent and structure. An event type can represent an event arriving from a producer or event produced by an event processing agent (see below).

We use an attribute-based event model, where an event type is a set of attributes and events are described by key-value pairs. Given an event e we define its *event type* by $e.type$. An event e is identified by a unique id $e.id$.

In the context of this work, we rely on existing models for monitoring business processes, such as the EPCglobal standard for RFID events [15]. Therefore, we define an event type for each activity in a process model. By $\mathbb{E}_P \subseteq \mathbb{E}$, we denote the set of all event types of a process model, e.g., $\mathbb{E}_P = \{a, \ldots, f\}$ for the process model in Fig. 2. An event of such an event type represents the occurrence of an activity in some process instance. Such an event is signalled at some point during the execution of the activity. Each event has an attribute $e.cid$ that refers to a specific process instance.

An *event processing agent* (EPA) is a processing element that applies logic to a set of input events, to generate a set of output (complex) events. A *context* is a named specification of conditions that groups event instances so that they can be jointly processed. While there exist several context dimensions, our work refers to two, namely the temporal context and attribute-based segmentation. A temporal context consists of one or more time intervals, possibly overlapping. Then, each interval is a context partition. As for the attribute-based segmentation, attribute cid is used for partitioning all events, so that event patterns (see below) always relate to events of a specific process instance. Each EPA is associated with an event context that, intuitively speaking, identifies what event instances are relevant for pattern detection, whereas the pattern itself defines the detection logic should be applied to these events.

An *event processing network* (EPN) [16] describes an event processing application as a set of EPAs, event producers, and event consumers, linked by channels. An *event channel* is a processing element that receives events from one or more source elements (producer or EPA), makes routing decisions, and sends the input events unchanged to one or more target elements (EPA or consumer) in accordance with the routing decisions. In an EPN, EPAs are communicating in asynchronous fashion by receiving and sending events.

3.2 Patterns: Syntax & Semantics

Pattern matching (also known as pattern detection) is a type of EPA that enables the analysis of collections of events and the relationship between them. Informally, we say that a conditional combination of events matches a pattern if this combination satisfies the particular pattern definition. Etzion and Niblett [14] identified a fixed set of pattern matching EPA types, each with well-defined logic. Here, we adopt a language that features *all* (conjunction), *any* (disjunction), and *seq* (sequence) patterns.

For each pattern matching EPA, three sets can be defined. A relevant event type set (RTS) of a complex event pattern is a list of event types on which a matching function is applied. A pattern participant set (PS) is a collection of event instances that occur within a pattern context partition (time window and cid); these events are instances of event types mentioned in pattern's RTS. The events $\{e_1, \ldots, e_n\}$ of a PS can be represented as an *event sequence* $\sigma = (e_1, \ldots, e_n)$, in which the order follows from the event timestamps, i.e., $e_i.t \leq e_{i+1}.t$ for $1 \leq i < n$. Finally, a pattern matching set (MS) is the output of the pattern matching process. It contains sets of events that appear in the PS of the same process and satisfy a certain pattern definition.

We define pattern EPAs recursively. First, an elementary pattern relates a set of event types with a certain operator to each other. Implicitly, such an elementary pattern defines another event type, a *compound* event type. We use $\mathbb{E}^*$ to denote the set of all finite sequences over event types and $|\mathcal{E}|$ for $\mathcal{E} \in \mathbb{E}^*$ to refer to the length of sequence $\mathcal{E}$.

Definition 2 (Elementary Pattern): An *elementary pattern* is a tuple $O = (\mathcal{E}(O), \delta(O))$ where
- $\mathcal{E}(O) \in \mathbb{E}^*$ is a finite sequence of event types,
- $\delta(O) \in \{all, seq, any\}$ is an operator type.

We overload set notations by applying them to sequences and write $E \in \mathcal{E}(O)$ if event type E is part of the sequence $\mathcal{E}(O)$.

As an example, consider the conjunction pattern $O_1 = ((a, d), all)$. Here, all the relevant event types ($\{a, d\}$) have at least one instance detected within the specific context and the events order is immaterial. As another example, $O_2 = ((a, d), seq)$ is a pattern that looks for at least one instance of each event type in a predefined order. The event sequence (d_1, a_1) where a_1 is of type a and d_1 is of type d satisfies the all pattern but not the seq pattern. Sequence (a_1, d_1) matches both patterns.

An elementary pattern O defines a compound event type, $O \in \mathbb{E}$. Hence, more complex pattern EPAs may be built by patterns over these event types.

Definition 3 (Pattern EPA): A *pattern EPA* is a tuple $(\mathcal{O}, \chi)$ where:
- $\mathcal{O}$ is a finite non-empty set of elementary patterns,
- $\chi : \bigcup_{O \in \mathcal{O}} \mathcal{E}(O) \rightarrow \mathcal{O}$ assigns event types to elementary patterns.

We further introduce a distinguished elementary pattern $\perp \in \mathbb{E}$ that never matches any event of any type. It can be seen as a placeholder that is later used in the definition of a pattern EPA for implementing pattern transformation.

In the following sections, we assume pattern EPAs

(1) to be closed: Event types are either primitive or refer to another elementary pattern, $\forall\ O \in \mathcal{O}, E \in \mathcal{E}(O)$ it holds that $E \in \mathbb{E}_P \cup \{\perp\}$ or $\exists\ O' \in \mathcal{O}$ with $\chi(E) = O'$.

(2) to be rooted: Each compound event type (except $\perp$) is referenced at most once, $\forall\ O_1, O_2, O_3 \in \mathcal{O}, E_1 \in \mathcal{E}(O_1), E_2 \in \mathcal{E}(O_2)$ with $\chi(E_1) = O_3$, $\chi(E_2) = O_3$, and $O_3 \neq \perp$ it holds $E_1 = E_2$. We use $\rho(O_3) = O_1 = O_2$ to denote the referencing pattern and define $\rho(O) = O$ if event type O is not referenced at all. The root compound event type $O_r \in \mathcal{O}$ is the only event type with $\rho(O_r) = O_r$.

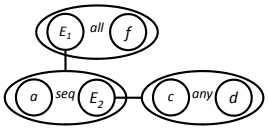

Fig. 3: Example of a pattern EPA, edges representing χ.

Figure 3 visualizes an exemplary pattern EPA built of multiple elementary patterns, formally:

$$\begin{aligned} Q_1 &= (\{O_1, O_2, O_3\}, \chi_1) \\ O_1 &= ((E_1, f), all); O_2 = ((a, E_2), seq); O_3 = ((c, d), any) \\ \chi_1 &= \{(E_1, O_2), (E_2, O_3)\}. \end{aligned}$$

Given a pattern $(\mathcal{O}, \chi)$, we use $\mathcal{E}(\mathcal{O}, \chi) = \{O_r\} \cup \bigcup_{O \in \mathcal{O}} \mathcal{E}(O)$ with O_r being the root of the pattern to refer to all primitive and compound event types of the pattern, i.e., the union of the RTSs. Further, we recursively define $\chi^* : \mathcal{E}(\mathcal{O}, \chi) \rightarrow \wp(\mathcal{E}(\mathcal{O}, \chi))$, with $\wp(\mathcal{E}(\mathcal{O}, \chi))$ as the power set of $\mathcal{E}(\mathcal{O}, \chi)$, to capture all event types needed for the definition of an event type, i.e., $E_2 \in \chi^*(E_1)$ if and only if $E_1 = E_2$ or if E_1 is a compound event type, $E_1 \in \mathcal{O}$, then $E_2 \in \chi^*(E_3)$ for some $E_3 \in \mathcal{E}(E_1)$.

Pattern semantics are defined as a function from a participant set (PS) to a pattern matching set (MS). The semantics depend on a set of processing policies, cf., [14]. We choose these policies against the background of detecting events that stem from business processes. That is, we assume that the event engine reports matches as soon as a pattern is evaluated to true (evaluation policy is *immediate*) and may reuse an event for creating multiple matching sets (consumption policy is *reuse*). Also, all matchings within an event context are considered (cardinality policy is *unrestricted*) and each matching event instance is reported (instance selection policy is *each*).

We define pattern semantics recursively. Let $\sigma = (e_1, \ldots, e_n)$ be an event sequence. For a primitive event type $E \in \mathbb{E}_P$, the participant set is defined as $PS(E) = \{e_1, \ldots, e_m\}$ with $e_i.type = E$ and $e_i.cid = e_j.cid$, $1 \leq i, j \leq m$. The matching set comprises m sets, each with a single event, i.e., $MS(E) = \{\{e_1\}, \ldots, \{e_m\}\}$. The distinguished event type $\perp$ has an empty matching set for any event sequence.

Each set $c = \{e_1, \ldots, e_k\} \in MS(E)$ of a matching set represents a compound event c with $c.type = E$ and $c.t = \max_{1 \leq i \leq k}(e_i.t)$ according to point-based semantics.

Semantics of an elementary pattern $O = (\mathcal{E}(O), \delta(O))$, therefore, is defined over the compound events matching the child event types. The participant set of O is defined as $PS(O) = \bigcup_{E \in \mathcal{E}(O)} MS(E)$. Note that the PS, again, induces an event sequence $(c_1, \ldots, c_m)$ in which the order follows from the timestamps of the compound events.

Given this event sequence $(c_1, \ldots, c_m)$, event types $\mathcal{E}(O) = (E_1, \ldots, E_l)$ and $\delta_O = all$, the MS of O is defined as

$$MS(O) = \{\{c_1, \ldots, c_l\} \subseteq PS(O) \mid c_i.type = E_i \text{ for } 1 \leq i \leq l\}.$$

If $\delta_O = seq$, the MS is defined as

$$MS(O) = \{\{c_1, \ldots, c_l\} \subseteq PS(O) \mid c_i.type = E_i \ \wedge\ e_1.t < \ldots < e_l.t \text{ for } 1 \leq i \leq l\}.$$

If $\delta_O = any$, the MS is defined as

$$MS(O) = \{\{c\} \subseteq PS(O) \mid c.type \in \mathcal{E}(O)\}.$$

Consider patterns O_1 and O_2 as defined earlier and event sequence (d_1, a_1, a_2). Then, we observe $MS(O_1) = \{\{d_1, a_1\}, \{d_1, a_2\}\}$ and $MS(Q_2) = \{\}$.

For a pattern EPA $(\mathcal{O}, \chi)$, semantics directly follows from the nesting of elementary patterns, each of them being evaluated over the sequence induced by the PS, which is derived from the compound events matching the child event types.

3.3 Execution Plans

Pattern execution can be captured by a state machine [17] where state transitions are triggered by event detections. A primitive event is realized by a two-state state machine, where the detection of an event results in a transition from the start state to the final (aka accepting) state. Patterns for complex events are built by concatenation through the merge of starting and accepting states of respective automata.

We rely on two extensions of the simple state machine based model. First, we constrain the evaluation along state transitions [18] and enforce the temporal and process instance context check along transitions. That is, state transitions are followed only for events that occur in the same time window and have equal values for attribute cid. The second extension extends the employed communication paradigm to be either push-based or pull-based, cf., [19]. Following a push strategy, the event engine subscribes to an event type and gets notified upon the arrival of a respective event. A pull-based strategy allows for pulling events of a particular type from a transaction log at a later stage.

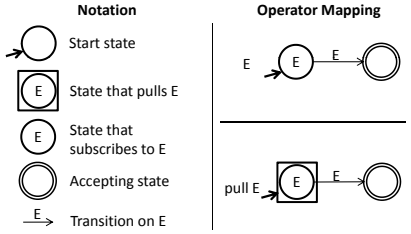

Fig. 4: Notation and plan formalization.

We are now ready to present *execution plans* as extended finite state machines (FSMs) [19]. The FSM has an active state for each partial detection of an event pattern. We use event types to denote an atomic plan and the plan operators $\rightarrow$ and $pull$ for building patterns of complex events. The notation and the mapping of plan operators are illustrated in Fig. 4.

TABLE 1: Execution plans for pattern operators

Pattern Operator	Alternative Sets of Execution Plans
$all(E_1, E_2)$	$\{E_1 \rightarrow E_2, E_2 \rightarrow E_1\}$, $\{E_1 \rightarrow \ pull\ E_2, E_2 \rightarrow \ pull\ E_1\}$, $\{E_2 \rightarrow E_1, E_2 \rightarrow \ pull\ E_1\}$, $\{E_1 \rightarrow E_2, E_1 \rightarrow \ pull\ E_2\}$
$seq(E_1, E_2)$	$\{E_1 \rightarrow E_2\}, \{E_2 \rightarrow \ pull\ E_1\}$
$any(E_1, E_2)$	$\{E_1, E_2\}$

The mapping of pattern operators to execution plans is illustrated for the case of binary operators in Table 1. For the all operator, two possible orderings need to be covered, each realized either as a pull-based or push-based plan. For the seq operator, only one ordering is addressed with either strategy. The choice whether to follow a pull-based or push-based execution is taken by the event engine. As part of our optimisation, we also consider this selection that may overrule the default choice if appropriate. Finally, for the any operator, two independent atomic plans have to be applied.

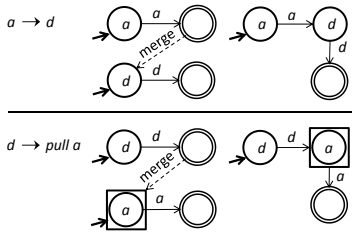

Fig. 5: Formalization of example plans.

For the example pattern $all(a, d)$, there exist four different sets of execution plans, each comprising two plans. For pattern $seq(a, d)$, in turn, either $a \rightarrow d$ or $d \rightarrow \ pull\ a$ is executed. Fig. 5 illustrates the realization of the pattern $seq\,(a, d)$ with automata. The top automaton detects an event of type a followed by the detection of an event of type d ($a \rightarrow d$) and the bottom one detects an event of type d, followed by pulling a ($d \rightarrow \ pull\ a$). Both automata are execution plans for this pattern. The first plan passively waits for events of type a and then for corresponding events of type d. The second plan waits for events of type d and subsequently actively pulls events of type a that happened before d. In both cases, the automata for the detection of a is merged with the one detecting d.

3.4 Ordering Event Occurrences

Recall that our setting involves events that indicate the execution of activities of a business process. Our optimisation of event processing leverages the fact that in a normative process model behavioural constraints are defined for activities, constraints that are propagated to events. As such, the behaviour of a process model, captured as a set of completed traces (see Section 2) can be transformed into valid event sequences that can be observed when executing the process. We employ the notion of a behavioural profile [8], [9] to capture constraints of a process model. In essence, such a profile defines three ordering relations between pairs of activities, namely strict order, exclusive execution, interleave ordering, and a co-occurrence relation to capture joint occurrences of activities.

Definition 4 ((Causal) Behavioural Profile): Let $P = (A, s,\ e, C, F, T)$ be a process model with $\mathcal{T}_P$ being its set of completed traces. A pair $(x, y) \in A \times A$ is in weak order, $x \succ y$, iff there exist a trace $(n_1, \ldots, n_m) \in \mathcal{T}_P$ with $n_j = x$ and $n_k = y$ for $1 \leq j < k \leq m$.

- A pair $(x, y) \in A \times A$ is in one of the following relations:
  - *strict order relation* $\rightsquigarrow_P$: $x \succ_P y$ and $y \not\succ_P x$.
  - *exclusiveness relation* $+_P$: $x \not\succ_P y$ and $y \not\succ_P x$.
  - *interleaving relation* $||_P$: $x \succ_P y$ and $y \succ_P x$.
- A pair $(x, y) \in A \times A$ is in the *co-occurrence relation* $\gg_P$, iff for all traces $(n_1, \ldots, n_m) \in \mathcal{T}_P$ it holds $n_i = x$ implies $n_j = y$ for $1 \leq i, j \leq m$.

$\mathcal{B}_P = \{\rightsquigarrow_P, +_P, ||_P\}$ is the *behavioural profile* of P. $\mathcal{B}_P^+ = \mathcal{B}_P \cup \{\gg_P\}$ is the *causal behavioural profile* of P.

The relations of the behavioural profile are pairwise distinct and, together with *reverse strict order* $\rightsquigarrow^{-1} = \{(x, y) \in A \times A \mid y \rightsquigarrow x\}$, partition the Cartesian product of activities A. Since the ordering relations do not define causality (an event is caused by another event), the causal behavioural profile adds a notion of co-occurrence. It holds if any trace that contains the first activity contains also the second activity. Note that the co-occurrence relation does not enforce a specific order, but only defines whether a complete trace containing an event of one type also contains an event of another type. We separate ordering from co-occurrence in the definition, since both aspects are considered at different stages of optimisation. Also, results on completeness are stated relative to either the behavioural profile or the causal behavioural profile.

Consider the event types for the activities of the process model in Fig. 2. It holds $a \rightsquigarrow c$, meaning that events of type a and c are ordered whenever they occur for the same process instance. Occurrence of an event of type a, however, does not imply the occurrence of an event of type c, whereas the reverse holds true. Hence, it holds $a \not\gg c$, but $c \gg a$. Further, $c + d$ and $a||b$ as events of both types never occur together or in any order for a single process instance, respectively.

The behavioural profile of a process model restricts the validity of event sequences that can be observed for any process instance. For the model in Fig. 2, for example, we may observe a sequence (s, b, a, d), which is in line with all constraints of the behavioural profile. Sequences (s, d, b, a) and (s, a, b, d, c) are not valid, since they violate $a \rightsquigarrow d$ and $d + c$.

For our notion of a process model, computation of the causal behavioural profile is done efficiently under the assumption of soundness. Soundness is a correctness criteria for process models that guarantees the absence of behavioural anomalies, such as deadlocks [20]. If a process model is sound the behavioural profile is computed in $O(n^3)$ time with n as the number of nodes of the graph [9]. If the process model satisfies certain assumptions on the structure of cycles, the causal behavioural profile is computed in $O(n^3)$ time as well.

4 Optimisation with Event Ordering

This section first gives an overview of our approach to optimisation based on process models (Section 4.1). Then, we discuss the different levels of optimisation (Sections 4.2 to 4.4) and outline the application of optimisation rules (Section 4.5).

4.1 Overview

We consider optimisation at three stages, see Fig. 6. First, we provide transformation rules that aim at reformulating the original pattern without changing its semantics in terms of the matching set. Second, we provide rules that guide the selection of an efficient execution plan for a given pattern. Finally, we provide rules that transform execution plans by specifying new events that enable more efficient pattern processing.

When designing the optimisation rules, we aim at three goals, namely correctness, efficiency, and completeness.

Correctness. Given a behavioural profile $\mathcal{B}_P^+$ and two patterns Q and Q', a transformation from Q to Q' is *correct* if all traces matched by Q and not matched by Q' cannot occur under $\mathcal{B}_P^+$, and all traces matched by Q' are matched by Q.

Efficiency. Efficiency can be measured in terms of *processed events* and *memory consumption*. Even though these measures are often not orthogonal, they provide an angle to decide on the application of optimisation that faces an inherent trade-off.

Reducing the number of processed events is relevant in scenarios where the network is a bottleneck or communication uses scarce resources. Since pattern transformation and plan selection do not introduce new event types, we identify improvements along this dimension based on the number of instantiated execution plans. Plan transformation adds event types to execution plans and, as such, never improves the number of processed events.

Reducing memory consumption of the event engine is relevant if resource constrained devices run the pattern or if intermediate results grow large. Analytically, improvements in memory consumption are approached from two angles. For pattern transformations and plan selection, a reduced number of instantiations of execution plans indicates less memory consumption. For plan transformation, we identify improved memory consumption by the time that events stay in the engine.

Completeness. Completeness of a set of transformation rules w.r.t. a behavioural profile ensures that no alternative set of rules can achieve further improvement to performance.

We use the notations introduced earlier and combine them in transformation rules as for-if-then statements, 'for $part_1$ if *condition*, then $part_2$'. Here, $part_1$ and $part_2$ can be patterns or partial execution plans. With each such statement, $part_1$ is transformed into $part_2$ if a condition on the pattern or execution plan structure and on the behavioural model is satisfied.

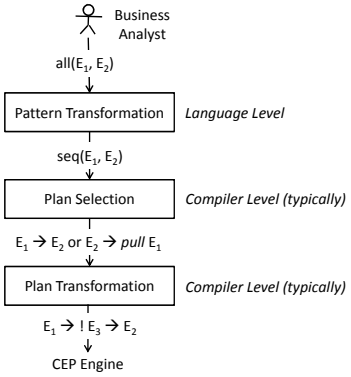

Fig. 6: Automatic tailoring of patterns to processes.

TABLE 2: Overview of pattern transformations

Pattern Operator	Process Knowledge: Strict Order	Rev. Strict Order	Exclusiveness	Interleaving
Rules for event types of a single elementary pattern				
all	seqs (Rule 1)	seqs (Rule 2)	faln (Rule 4)	—
seq	faln (Rule 3)	faln (Rule 3)	faln (Rule 4)	—
any	—	—	—	—
Rules for event types of independent elementary patterns				
all	—	—	—	—
seq	—	—	—	—
any	faln (Rule 5)	faln (Rule 5)	faln (Rule 6)	—
Rules for pruning falsified event types				
all	prn (Rule 7)			
seq	prn (Rule 7)			
any	prn (Rules 8/9)			

4.2 Pattern Transformation

Given a pattern and a process model, pattern transformation modifies the *pattern definition* in a way that maintains correctness and improves performance. We consider rules for pattern transformation that are *local*, i.e., changing a single elementary pattern of a pattern EPA. Table 2 summarizes the set of considered transformations. For each of the pattern operators (all, seq, any), we consider three cases: a behavioural relation refers to (1) event types that are required to define an elementary pattern, (2) independent elementary patterns, which are both part of the definition of a pattern EPA, but are independent in the sense that neither of them is required for the definition of the other, (3) auxiliary rules that simplify the definition of a pattern EPA based on the transformations done as part of first two cases. For all cases, we consider all behavioural relations. For rules that are conceptually close to each other, we present a formalization of one rule. Formalisations of additional rules are available in a supplement [21].

We use three types of justification to motivate the use of a rule. The first, *sequentialization* (abbreviated as seqs), enforces a specific ordering of events. The second, *falsification* (abbreviated as faln), falsifies certain event types. The third, *pruning* (abbreviated as prn), simplifies a pattern by removing event types that have been falsified.

Single Elementary Patterns. We start with the rules that consider event types that are part of the definition of a single elementary pattern. Consider the all operator in the presence of order constraints. An order constraint between event types is leveraged for optimisation by transforming an all operator into a seq operator to consider only traces showing the ordering induced by the process model. Given a pattern $all(E_1, E_2)$ and $E_1 \rightsquigarrow E_2$, intuitively, any matching event of a certain type E_2 is preceded by all corresponding events of type E_1. By applying the rule, we avoid receiving and storing events of type E_2 that have no matching event of type E_1.

Rule 1 :
for $(\mathcal{O}, \chi)$
if $\exists\, O = ((O_1, \ldots, O_n), \delta(O)) \in \mathcal{O}, \delta(O) = all$
and $\forall\, E_1, E_2 \in \mathbb{E}_P, E_1 \in \chi^*(O_i), E_2 \in \chi^*(O_j), 1 \leq i < j \leq n :$
$E_1 \rightsquigarrow E_2$
then $(\mathcal{O} \cup \{O'\} \setminus \{O\}, \chi')$
$O' = ((O_1, \ldots, O_n), seq)$
$\chi' = \{(E, E') \in \chi \mid E' \neq O\} \cup \{(\rho(O), O')\}$

Rule 2 adapts rule 1 for the reversed ordering of events in

which event types E_1 and E_2 are related by reverse strict order. As an example, consider our initial process model (Fig. 2) and the pattern $all(f, all(a, d))$. Since the process model induces $a \rightsquigarrow d, a \rightsquigarrow f$, and $d \rightsquigarrow f$, the pattern is transformed into $seq(seq(a, d), f)$ by applying rules 1 and 2 in either order.

Order constraints are also leveraged for optimising a seq pattern. If the requested ordering of events contradicts with the one induced by the process model, the resulting compound event type is falsified by replacing it with the distinguished type $\perp$ that does not match any events. Hence, the rule prevents attempts to match event combinations that cannot occur.

Rule 3 :
for $(\mathcal{O}, \chi)$
if $\exists\, O = ((O_1, \ldots, O_n), \delta(O)) \in \mathcal{O}, \delta(O) = seq$
and $\exists\, E_1, E_2 \in \mathbb{E}_P, E_1 \in \chi^*(O_i), E_2 \in \chi^*(O_j), 1 \leq i < j \leq n :$
$\forall\, O' \in \chi^*(O_i) \setminus \mathbb{E}_P, E_1 \in \chi^*(O') : \delta(O') \in \{all, seq\} \wedge$
$\forall\, O' \in \chi^*(O_j) \setminus \mathbb{E}_P, E_2 \in \chi^*(O') : \delta(O') \in \{all, seq\} \wedge$
$E_1 \rightsquigarrow^{-1} E_2$
then $(\mathcal{O}', \chi')$
$\mathcal{O}' = \mathcal{O} \cup \{\perp\} \setminus \{O\} \setminus \{O' \in \mathcal{O} \mid \forall\, (E, E') \in \chi' : O' \neq E'\}$
$\chi' = \{(E, E') \in \chi \mid E \notin \chi^*(O)\} \cup \{(\rho(O), \perp)\}$

Since (reverse) strict order is irreflexive and antisymmetric, rule 3 implicitly covers both cases as indicated in Table 2.
A similar optimisation is achieved by exploiting exclusiveness. Rule 4 falsifies compound event types (of all or seq patterns) if at least two of the referenced primitive event types are exclusive and referenced by all and seq operators only.

Independent Elementary Patterns. So far, we proposed rules that, given an elementary pattern, leverage information about two primitive event types required (directly or indirectly) to define the respective pattern. However, conclusions may also be drawn for two primitive event types, such that only one of them is part of the definition of an elementary pattern, whereas the other event type is independent of this pattern but referenced by another pattern in the same pattern EPA.

The next rule considers a primitive event type E_1 used to define an any pattern. It requires that $E1$ is mandatory for one event type E of the any pattern, i.e., matching an event of type E relies on the occurrence of an event of type E_1 in all cases ($E = E_1$ or E_1 is referenced by all and seq operators). Then, the any pattern may be required (directly or indirectly) to define a seq pattern. Now, we consider a primitive event type E_2 that is mandatory for the definition of the seq pattern, such that the respective events must occur after the events matching the any pattern. If we observe an order between events of those primitive types that is not in line with the order in the seq pattern, i.e., $E_1 \rightsquigarrow^{-1} E_2$, the respective event type in the any pattern is falsified. Hence, we limit the any pattern by falsifying one of the alternatives of the disjunction over (primitive or compound) event types.

Rule 5 :
for $(\mathcal{O}, \chi)$
if $\exists\, O = ((O_1, \ldots, O_n), \delta(O)) \in \mathcal{O}, \delta(O) = any \wedge$
$\exists\, O' = ((O'_1, \ldots, O'_m), \delta(O')) \in \mathcal{O}, \delta(O') = seq,$
$O \in \chi^*(O'_i), 1 \leq i \leq m$
and $\exists\, E_1, E_2 \in \mathbb{E}_P, E_1 \in \chi^*(O_j), E_2 \in \chi^*(O'_k),$
$1 \leq j \leq n, 1 \leq k \leq m, i < k :$
$\forall\, O'' \in \chi^*(O_j) \setminus \mathbb{E}_P, E_1 \in \chi^*(O'') : \delta(O'') \in \{all, seq\} \wedge$
$\forall\, O'' \in \chi^*(O'_k) \setminus \mathbb{E}_P, E_2 \in \chi^*(O'') : \delta(O'') \in \{all, seq\} \wedge$
$E_1 \rightsquigarrow^{-1} E_2$
then $(\mathcal{O}', \chi')$
$\mathcal{O}' = \mathcal{O} \cup \{\perp\} \setminus \{O_j\} \setminus \{\hat{O} \in \mathcal{O} \mid \forall\, (E, E') \in \chi' : \hat{O} \neq E'\}$
$\chi' = \{(E, E') \in \chi \mid E \notin \chi^*(O_j)\} \cup \{(\rho(O_j), \perp)\}$

TABLE 3: Overview of plan selections

Pattern Operator	Process Knowledge for Event Types E_1 and E_2: Co-occur. either	No co-occur.	Co-occur. E_2 to E_1	Co-occur. E_1 to E_2
all	—	—	—	—
seq	—	—	Pull (Rule 10)	Push (Rule 11)
any	—	—	—	—

As for rule 3, also rule 5 covers the case of (reverse) strict order since the relations are irreflexive and antisymmetric.
Again, rule 5 that falsifies based on contradicting order may be adapted to exploit exclusiveness. This is realized by rule 6, that falsifies an alternative of an any pattern that will never contribute to matches of the complete pattern EPA since it requires matching events of a type that is exclusive to another type that is mandatory for matching the whole pattern. As an example, consider the pattern $any(all(c, seq(b, d)), f)$. Since our initial process model (Fig. 2) induces $c+d$, $all(c, seq(b, d))$ is falsified by rule 6, resulting in the pattern $any(\perp, f)$.

Pruning. Several of the rules falsify event types by introducing the type $\perp$. The last set of pattern transformation rules incorporate these falsified event types and prune the pattern definition. For all and seq operators, the joint occurrence of events of the referenced types is required. Hence, having a single referenced type falsified leads to pruning of the complete elementary pattern. In case the pattern was referenced by another pattern, the respective event type is falsified.

Rule 7 :
for $(\mathcal{O}, \chi)$
if $\exists\, O = ((O_1, \ldots, O_n), \delta(O)) \in \mathcal{O}, \delta(O) \in \{all, seq\},$
$O_i = \perp, 1 \leq i \leq n$
then $(\mathcal{O}', \chi')$
$\mathcal{O}' = \mathcal{O} \setminus \{O\} \setminus \{O'' \in \mathcal{O} \mid \forall\, (E, E') \in \chi' : O'' \neq E'\}$
$\chi' = \{(E, E') \in \chi \mid E \notin \chi^*(O)\} \cup \{(\rho(O), \perp)\}$

Finally, handling of falsified event types is also addressed for any patterns by two rules. Rule 8 prunes the falsified event type from the definition of an any pattern. Rule 9 prunes complete any patterns that are defined over a single falsified event type. With these rules, the aforementioned example pattern $any(\perp, f)$ is transformed into $any(f)$.

Having introduced a set of optimisation rules for plan transformation, we are able to conclude on their correctness, efficiency, and completeness for local pattern transformation given the outlined spectrum of pattern operators and process knowledge. Due to space limitations, the respective proofs can be found in the supplement [21].

Theorem 1: Rules 1-9 are correct, improve pattern processing, and are complete for all, seq, and any, under $\mathcal{B}_P$.

4.3 Plan Selection

Rules for plan selection are applied in the process of execution plan generation. The plan generation selects execution plans for all primitive and compound event types in isolation. The rules for plan selection remove inefficient execution plans from a set of candidate plans based on the co-occurrence relation of the causal behavioural profile. The pattern operators and the combinations of co-occurrence dependencies induce a spectrum that is explored for optimisation, as summarized in Table 3.

TABLE 4: Overview of plan transformations

Execution Plan	Process Knowledge: Exclusive only to first	Exclusive only to second	Intermediate event
$pull$-based	—	—	—
$push$-based	—	Early Term. (Rule 12)	Postp. Pull (Rule 13)

Optimisation based on co-occurrences refers solely to the seq operator. However, an all operator may be transformed into a seq operator using rules 1 and 2 . Plan selection decides on either push or pull strategy (cf., Section 3.3), thereby overruling the default strategy to achieve efficient processing.

Plan selection based on co-occurrence applies, once we look for (sets of) events in a sequence and observe that the occurrence of the second event (or set of events) implies the occurrence of the first event (or set of events). Then, an execution plan following a pull-strategy is beneficial. Assume that we derive from the behavioural profile that each event of type E_2 matches an event of type E_1, but not vice versa. The event of type E_1 may happen more often than those of type E_2. Pulling E_1 avoids processing irrelevant events of type E_1.

Rule 10 :
for $(\mathcal{O}, \chi)$
if $\exists\ O = ((O_1, \ldots, O_n), \delta(O)) \in \mathcal{O}, \delta(O) = seq,$
and $\forall\ E_1, E_2 \in \mathbb{E}_P, E_1 \in \chi^*(O_i), E_2 \in \chi^*(O_{i+1}), 1 \leq i \leq n :$
$E_1 \not\gg E_2 \wedge E_2 \gg E_1$
then $O_{i+1} \rightarrow\ pull\ O_i$

For illustration, consider the pattern $seq(seq(a, d), f)$ discussed above. Our initial process model induces $a \gg f, d \gg a, d \gg f$, and $f \gg a$. Thus, rule 10 suggests to rely on a pull-based plan for the expression $seq(a, d)$ ($d \rightarrow\ pull\ a$). For the second operator (the root of the pattern), we would not suggest any plan, but rather rely on the default strategy of the event engine.

The mirrored case of rule 10, i.e., co-occurrence from the first event type to the second one in a sequence, is captured by rule 11. This rule suggests a push-based execution plan.

Plan selection rules choose between valid plans and, thus, correctness is not an issue here. Yet, we are able to conclude on the efficiency and completeness of the presented rules for plan selection (proofs can be found in the supplement [21]).

Theorem 2: Rules 10-11 improve pattern processing and are complete for all, seq, and any under $\mathcal{B}_P^+$.

4.4 Plan Transformation

The last phase of optimisation transforms execution plan by adding event types to patterns. Plan transformation rules allow balancing two optimisation goals, namely the number of processed events and memory consumption. Detection of additional events increases the number of processed events, yet reduces memory consumption by allowing early termination or delayed loading of events into memory.

Given an execution plan for two event types, incorporating an additional event type is beneficial if it can differentiate among different ordering of events. We exploit exclusiveness and strict order for that purpose. We consider the case where an additional event type is exclusive only to the first event type or the second event type or that its occurrence lies between the event types of the plan in terms of strict order. The spectrum investigated for optimisation is spanned by these cases and the strategies followed by an execution plan, see Table 4. Using the behavioural profile, we either decide to *early terminate* the processing or to *postpone pull* operations.

The first rule is applicable for a push-based plan that features two event types, if there exists an event type X that is in strict order with the first type (or all primitive event types) and exclusive to the second event type (or one of the primitive event types). The absence of an occurrence of an event of type X is a necessary condition for matching the original plan. Hence, the occurrence of an event of this type may terminate execution of the plan.

Rule 12 :
for $(\mathcal{O}, \chi)\ \wedge\ O_1 \rightarrow O_2$
if $\exists\ X \in \mathbb{E}_P :$
$\forall\ E \in \mathcal{E}_P, E \in \chi^*(O_1) : E \rightsquigarrow X\ \wedge$
$\exists\ E \in \mathcal{E}_P, E \in \chi^*(O_2) : X + E$
then $O_1 \rightarrow \neg X \rightarrow O_2$

The occurrence of an event of type X indicates that there is no matching event of type O_2. Thus, the event of type O_1 can be dropped. Even though the rule increases the number of processed events, it reduces the memory consumption. Consider the plan $b \rightarrow c$. Based on the behavioural profile of our initial process model (Fig. 2), there is an event type d and it holds $b \rightsquigarrow d$ and $d + c$. Hence, rule 12 is applicable and the execution plan optimised for memory consumption is $b \rightarrow \neg d \rightarrow c$.
A symmetric rule for optimising pull-based plans, $O_1 \rightarrow\ pull\ O_2$, does not have a clear-cut performance improvement. Consider an event type X that is exclusive to O_2 and in strict order with O_1. Even though X signals that the plan may be dropped, such a rule would not be beneficial in the general case. Although the memory needed to keep an event of type O_1 while pulling an event of type O_2 can be saved, for a concrete setting this effort may be negligible. Therefore, the increased number of processed events cannot be justified without further knowledge on event frequencies.

Finally, an additional event that occurs between the events referenced in the execution plan may be leveraged for optimisation. If the occurrence of such an intermediate event is implied by the first event, and implies the occurrence of the second event, it may be used to trigger pulling of an event of the first type at a later stage. The rule is applicable for both push-strategies and pull-strategies.

Rule 13 :
for $(\mathcal{O}, \chi)\ \wedge\ O_1 \rightarrow O_2$
if $\exists\ X \in \mathbb{E}_P :$
$\forall\ E \in \mathcal{E}_P, E \in \chi^*(O_1) : E \gg X \wedge E \rightsquigarrow X\ \wedge$
$\forall\ E \in \mathcal{E}_P, E \in \chi^*(O_2) : X \gg E \wedge X \rightsquigarrow E$
then $X \rightarrow\ pull\ O_1 \rightarrow O_2$

Pulling an event of type O_1 is performed once an event of type X, indicating that an event of type O_2 is about to happen, is detected. This way, O_1 is available when O_2 is detected, thereby reducing memory consumption of the event processor.

As for rule 12, there exists also a symmetric rule for rule 13 that optimises pull-based plan. Without specifically considering the potentially different costs for realizing pull- or push-based plan execution, such a rule is not justified. According to our analytically model, a clear-cut performance improvement is achieved only for the push-based plan.

Again, we can show correctness of the presented plan transformation rules (proofs can be found in the supplement [21]). In terms of efficiency, both rules insert additional event types thereby increasing the number of processed events, but reduce the memory consumption in the event processor. Without further assumptions on the implementation of a pull operator, however, conclusions on the completeness of the presented plan transformation rules cannot be drawn.

Theorem 3: Rules 12-13 are correct and improve memory consumption.

4.5 Rule Application

Apart from the levels of pattern processing, i.e., pattern transformation, plan selection, and plan transformation, there are causal dependencies between rules. That is, the application of a rule may enable or disable the application of other rules. Rules 1 and 2 rewrite an all operator to a seq operator, which may enable rule 3. Further, rules 3-6 disable rules 1 and 2, since they falsify event types. Falsification, however, may enable rules 7-9 for pruning event types. With respect to the rules for plan selection, rules 1 and 2 may enable rule 10 or rule 11. Rules 3 to 6 may avoid plan generation. For falsified event types there is no need to apply plan selection rules. There are further dependencies between the rules for plan selection and plan transformation. Rule 10 enforces a pull-strategy, which disables the application of the plan transformation rules 12 and 13. Enabling of rules 12 and 13 also depends on the default communication strategy of the event engine.

Based on the discussed dependencies, the general scheme for the application of the rules is illustrated in Fig. 7. That is, rewriting by rules 1 and 2 is applied first, followed by rules 3-6 for falsification. Subsequently, pruning by rules 7-9 is applied. Then, plan selection is based on rules 10 and 11. Only if reducing memory consumption is the primary optimisation goal, we leverage rules 12 and 13 for plan transformation.

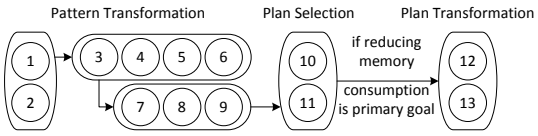

Fig. 7: Scheme for the application of optimisation rules.

5 Evaluation

To evaluate our approach, we experimented with a large case from industry to answer the following questions:

- What is the applicability of our optimisation rules and what are the achieved savings for real-world processes?
- To which extent do these savings materialise in systems for event processing?
- What improvements can be expected when combining the proposed method with existing optimisation techniques?

Below, we first give background on the industry case and the dataset (Section 5.1). Then, we turn to an analytical evaluation of the applicability of optimisation rules and the savings obtained for the given case (Section 5.2). To investigate how the analytically determined savings materialised in specific systems, we report on experiments with the Streams framework and the Esper engine (Section 5.3). Section 5.4 investigates the joint application of our technique and other optimisations, before we reflect on the evaluation results in Section 5.5.

5.1 Background & Dataset

Our evaluation is based on more than 1000 processes of an insurance company that focuses on health insurances. The organisation comprises more than 5000 employees and shows a high level of process-orientation. That is, the majority of operations, such as negotiation of insurance policies and claim handling, are described by detailed business process models.

Processes. As part of this case study, we were granted access to 1021 process models, which are not publicly available due to non-disclosure agreements. The average size of the models in the collection is around 23 nodes with a maximum of 339 nodes. The models show a high quality in terms of syntax and semantics. Only one model has a syntax error and four models are not sound [20], i.e., they contain behavioural anomalies such as deadlocks. These five models have been excluded from our evaluation, which leaves a set of 1017 process models.

For these processes, event logs could not be disclosed due to legal regulations and privacy considerations. However, the provided models include annotations about the average duration of all activities and about branching probabilities at all decision points in a fine-granular manner. Most activities need around 0.5 to 4 time units for completing, rather abstract activities take up to 876996 time units. Branching probabilities range from 0.003 to 0.995. Based on this information, we were able to simulate realistic event streams for the purpose of evaluation. To achieve robust results, we simulated 1000 instances per process, i.e., a total of 1,017,000 process instances. For each process, this created up to 83,000 events.

Event Query Patterns. For the insurer, monitoring execution time between process activities is of particular importance. On the one hand, this information is required to ensure that SLAs with external partners are not violated. On the other hand, many internal key performance indicators (KPIs) are based on these times. Against this background, we consider two collections of event query patterns for our experiments.

Decision Patterns relate to the execution of a process until a decision is taken. These patterns match a sequence of event types that refer to an activity indicating the start of the process and an activity directly following a decision point. The concrete set of such monitoring patterns could not be revealed by the insurance company, so we extracted all candidate patterns from the process models, leading to a set of 5636 event patterns.

Full Baseline Patterns relate to a sequence of activities that is executed within the same process instance (i.e., the time between the executions of these activities is important). This scenario is captured by sequential event patterns that relate of a certain length. Since, again, the actual monitoring patterns could not be revealed, we opted for considering all pairwise candidates as a baseline. For our model collection, this resulted in 373819 event patterns which include the first collection.

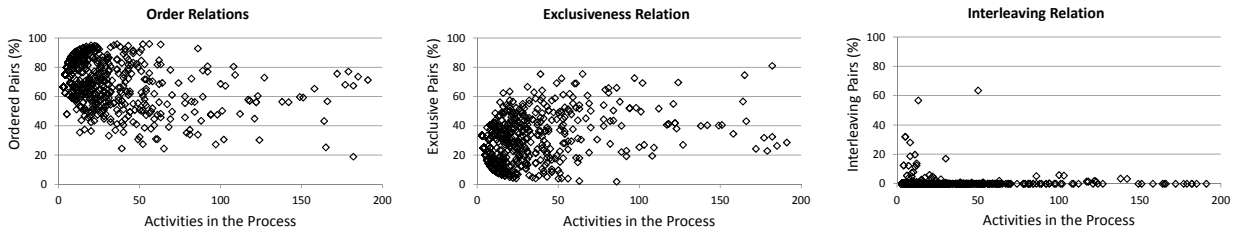

Fig. 8: Relation sizes relative to the number of activities of the process.

5.2 Analytical Evaluation

In this section, we address the questions of the applicability of our approach and of the achieved savings. For the rules for pattern transformation (rules 1 to 9), we focus on the potential applicability in terms of the exploitable process knowledge. The reason for that is that our pattern collections are motivated by the use case, but have been generated, whereas the actual benefit of pattern transformation highly depends on the patterns defined by an analyst. Plan selection and plan transformation, in turn, may well be evaluated with our pattern collections since the exact way of defining these queries has a minor impact on the resulting execution plans. For instance, an analyst may define decision patterns individually (separate EPAs as in our collection) or create a combination thereof (e.g., an EPA is defined as a sequence of the process start followed by a disjunction over the activities succeeding a decision point). Clearly, this influences the applicability of pattern transformation, whereas the set of required execution plans remains unchanged.

Setup. We approach potential applicability based on the observation that not all process constraints are equally usable for pattern transformation (Table 2). Thus, the sizes of the (reverse) strict order and exclusiveness relations that can be exploited indicate potential applicability of pattern transformation.

For evaluating plan selection, we assume a push-based strategy by default, which is in line with most CEP engines. Hence, we evaluate rule 10, which, if applied, overrides the default processing by suggesting a pull-based plan. In addition to counting how often the rule is applicable for a pair of event types in the event pattern, we also generated a plain and an optimised version of the pattern when a rule applies. The comparison of the efficiency of both plans allows for judging the achieved savings. Evaluation of plan transformation, namely rule 12 and rule 13, follows the very same approach. Rule 13 further requires selection of an event type X to be added to the execution plan. In our evaluation, we used a disjunction of all candidate event types, which is the worst case assumption.

Processing efficiency is measured based on the aforementioned analytic model (Section 4.1). Since memory consumption of a pull operation is dependent on the implementation of an event engine, we relied on the number of plan instantiations for pull-based execution plans when evaluating rule 10. For push-based plans (rules 12 and 13), we measured the precise time that events remains in memory to quantify memory consumption.

Results. Figure 8 illustrates the relative sizes of the behavioural profile relations. Each data point represents the percentage of activities that are covered by the respective relations of a single process. The majority of activity pairs is ordered. For large models (>50 activities), order relations cover between 40% and 80% of activity pairs. The remaining pairs are mostly covered by exclusiveness. For large models, exclusiveness relates to a significant share of activity pairs, around 20% to 80%, in nearly all models. Only a very few, rather small models have a large interleaving relation. With only a very few activity patterns being part of the interleaving relation that cannot be used for optimisation, our results point at a high potential applicability of pattern transformation.

Next, we turn the focus on the applicability of the rules for plan selection (rule 10) and plan transformation (rules 12 and 13). Rules 10 and 12 are applicable for virtually all of the *decision patterns*, i.e., for 95.40% and 98.99% of the patterns. Rule 13 is not applicable because of the decision point that directly precedes the activities considered in the *decision patterns*. The results for the *full baseline patterns* are shown in Fig. 9. Rules 10 and 12 are applicable in all processes with more than 24 activities and apply to a large share of event type pairs. For rule 12, applicability increases with the size of the processes and reaches over 94% for the largest process. For plan transformation with rule 13, in turn, the proportion of optimisable pairs decreases with the size of the processes. Still, rule 13 applies to more than 37% of all pairs over the whole sample.

The distribution of the average saving achieved by the rules for both pattern collections is illustrated in Fig. 10. For plan selection with rule 10 we observe high average reduction of plan instantiations, 64% (*decision patterns*) and 45% (*full baseline patterns*) on average, indicating high effectiveness of this rule. Good results are also obtained for plan transformation with rule 12 with improvements for 52% and 45% of the pairs, respectively. Pattern transformation with rule 13, applicable only for the *full baseline patterns*, yields less savings than rule 10 and rule 12. However, for 47% of the processes we still achieve savings over 20%. Assessing the relation between achieved savings and the size of the processes, the trends from the applicability analysis are largely confirmed, i.e., rule 10 shows no trend, whereas savings increasing with size are observed for rule 12, and savings decrease for rule 13 (the plots can be found in the supplement [21]).

5.3 Empirical Performance Analysis

The analytical model followed above makes the evaluation independent of any specific implementation. To answer the question to which extent these savings actually materialise in state-of-the-art systems, we implemented two sample applications. The first one is based on Streams (version 0.9.9), an

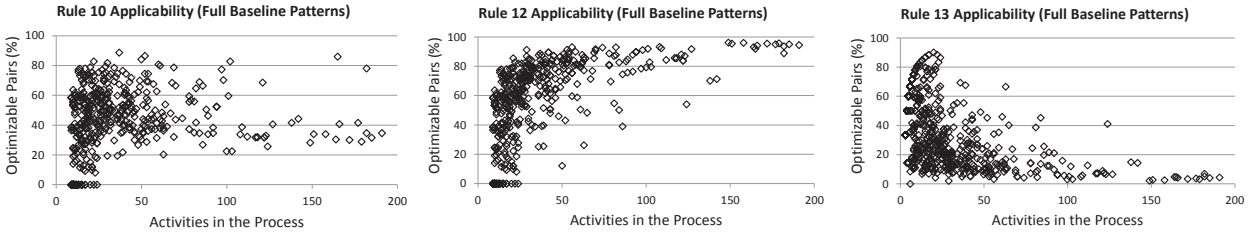

Fig. 9: Applicability of the rules for plan selection and plan transformation for the full baseline patterns.

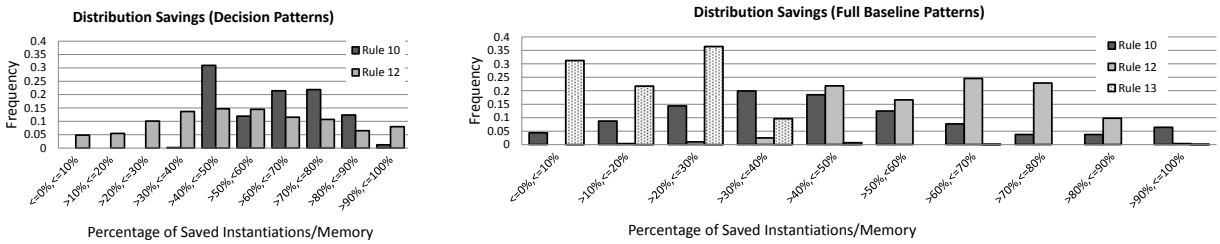

Fig. 10: Distribution of savings achieved with rules 10, 12, and 13 for both pattern collections.

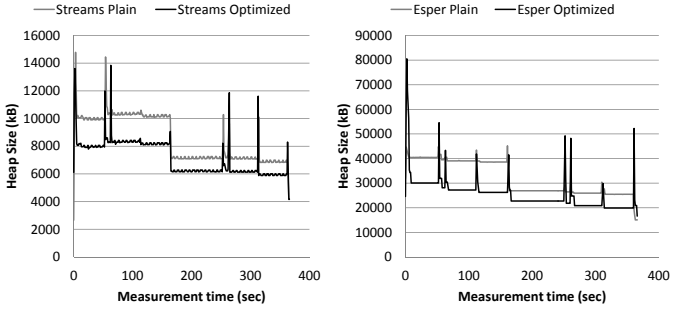

Fig. 11: Memory usage in two sample applications.

open source framework for stream processing grounded on data flow graphs, which allows for evaluation in a rather low-level framework. Our second implementation, in turn, targets the application of our approach for a common CEP engine. Here, we relied on the Esper engine (version 4.9), which is publicly available, to implement the event query patterns.

Setup. We tested both implementations with a process that has a suitable duration for real time simulation (63 possible pairs of event types) and ran the *full baseline patterns*. As an optimisation, we applied pattern transformation according to rule 12, which determines sets of events that allow for early termination of registered patterns. Out of the 63 possible pairs of event types, 26 may be optimised using this rule.

Results. Figure 11 shows the resulting heap utilization for both implementations, when all optimisable patterns are registered in parallel and garbage collection is forced every second. Note that the figures show the complete heap for the sample applications and not only the event memory. According to our analytical evaluation, 66% of the event memory can be saved. Including the overhead of the event forwarding and monitoring application, we observed about 16% (Streams) and 20% (Esper) reduction in heap utilization in our experiment. As such, the savings that materialize under realistic conditions are smaller than what can be expected from an analytical evaluation. However, we also see that significant savings are still achieved using both implementation approaches. For the case of the Esper engine, this is particularly remarkable since optimisation is added on top of the engine using additional rules. Hence, integration of the optimisation approach in the CEP query processor is likely to lead to further improvements.

5.4 Evaluation of Combined Optimisation

By exploiting the information given by a process model, our optimisation is largely independent of existing, general purpose optimisation techniques. In this experiment, we investigate this aspect for a sample application, which, again, was implemented in Streams (version 0.9.9). We relied on the execution model and optimisations proposed by Wu et al. [18]. They first provide a simple to program but inefficient model for implementing execution plans based on state machines, which is optimised using active instance stacks, stack partitioning over event contexts, and a strategy for operator placement to reduce intermediate result sizes. We implemented this model and the optimisations in the Streams application.

Setup. For evaluation, we used the same process as in the previous experiment. To increase the load placed on the system, we created 100 distinct execution plans as follows. Each plan implements a concatenation of four *full baseline patterns*. We ensured that this concatenation refers to a valid execution sequence of the process. Again, rule 12 for pattern transformation was used for optimisation.

Results. Figure 11 shows the resulting heap utilization, on a logarithmic scale, of the application running with either the simplistic (Naïve) or optimised (Advanced) execution model defined in [18], potentially combined with our optimisation technique (Process Opt.). For the simplistic model, activating our optimisation drastically reduces memory consumption by 59% overall. Here, events cannot be processed instantaneously, so that, as a side effect, the memory savings also lead to a significant speed up of the application (finishing after 14 min instead of more than 21 min). Using the optimised model by

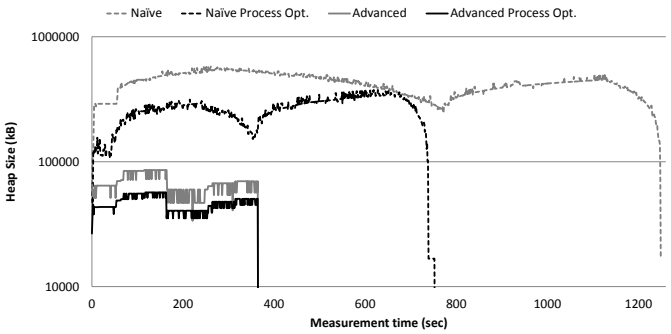

Fig. 12: Memory usage with different optimisations.

Wu et al. [18] greatly reduces memory consumption and allows for instantaneous processing. Yet, activating our optimisation technique yields significant memory savings of 29% overall.

5.5 Discussion

Our evaluation spanned various dimensions, among them the size and complexity of the processes, different sets of event query patterns, the realisability of improvements in state-of-the-art systems, and potential combination with other optimisation techniques. However, applicability and achievable savings clearly depend on the characteristics of the considered processes. Although the considered 1017 processes cover a broad range of divisions within the company, they still stem from a single company. We, therefore, performed the analytical evaluation also for another dataset, a publicly available event log of a paper review process. The results from this experiment largely confirm the observations done for the case presented here and can be found in the supplement [21].

Nevertheless, conclusions for the general case should be drawn cautiously. For instance, the processes show a rather low degree of parallelism and loops, such that most models have a small or empty interleaving relation. Also, the size of processes is not evenly distributed within the collection, which contains more smaller than larger processes. Though, this distribution is reflective of other process model collections, see [23].

Despite those limitations, our evaluation indicates that significant savings can be obtained by leveraging process knowledge for conducting event pattern matching. Also, the applicability results illustrate that optimisation is not limited to few patterns. Thus, one can expect benefits without strong assumptions on the selection of query event types.

6 Related Work

In the last decade, various complex event processing systems have been developed, among them Amit [3], SASE [18], Cayuga [24], CEDR [25]. These systems allow for defining event pattern EPAs as considered in this work and may be applied over event streams that originate from the operation of business processes, see [26], [27]. More specifically, our work is related to the optimisations of complex event processing, the specification of event patterns, behavioural verification and compliance checking, and transactional workflows.

In classical database research, semantic query optimisation aims at guiding the transformation of queries and the selection and transformation of execution plans [28], [29]. In the same way, our approach targets optimisation of the different phases of event pattern detection, yet with a behavioural focus. As much as semantic query optimisation exploits information on the schema level, our work relies on constraints defined by a normative process model for event types.

Many works on pattern optimisation address the application domain of wireless sensor networks (e.g. [30], [31]). Solutions for this domain mainly aim at reducing network load by pushing pattern operators close to the event sources. Other work presents general purpose approaches to pattern optimisation. Srivastava et al. optimise execution plans under consideration of differences in the capabilities of available devices [32]. Moreover, network delays can be considered in finding optimal operator placements and corresponding execution plans [33]. Pattern execution plans can also be rewritten to minimize resource consumption [34]. Akdere et al. combine push- and pull-based strategies to optimise execution plans [19]. Our approach takes up these ideas when leveraging process knowledge to decide for a push- or a pull-strategy. Wu et al. [18] suggest various optimisations for pattern matching, among them the usage of active instance stacks, stack partitioning and placement of operators to reduce intermediate result sizes. Such optimisations are largely orthogonal to our approach and, thus, can be combined. In our evaluation, we demonstrated the benefits of such combination when combining the techniques by Wu et al. [18] with our approach.

Ideas towards specifically leveraging knowledge of the business process have been discussed in some prior works. A general methodology for pattern rewriting, which is similar in spirit to the first stage of pattern transformation, was proposed by Rabinovich et al. [35]. In this work, we show how pattern rewriting can be instantiated in the presence of business processes. Close to our work are the approaches of Ding et al. [36] and BP-Mon by Beeri et al. [37], which also rely on knowledge about a business process to optimise event processing. Ding et al. [36] define static and dynamic satisfiability problems for event patterns against order and occurrence constraints that are derived from a process model. As such, the approach is similar to the proposed plan transformation rules that lead to negation of a pattern, but does not involve plan selection and plan execution. In contrast to our work, the extraction of constraints from a process model is left open there. Further, static and dynamic satisfiability can be solved efficiently only for conjunctive patterns (all and seq), whereas our approach allows for efficient handling of patterns including disjunction (any). BP-Mon [37] defines a pattern specification language that adopts process modelling concepts instead of relying on a standard CEP model. Patterns are behavioural scenarios, so that pattern matching is traced back to behavioural containment with the process specification. As an optimisation, BP-Mon considers relevance and inconsistency of activities w.r.t. an event pattern, which is extracted heuristically based on queries against the process model structure. Our work goes beyond this approach by not only considering structural features, but grounding the optimisation in behavioural relations. Also, the BP-Mon

optimisations are limited to identifying non-matches, whereas we exploit processes also for selecting among execution plans and transforming such plans to improve processing efficiency.

In our approach, we define automatic rules for deriving optimisation patterns from a normative process model. The specification of such event patterns typically builds on formal languages, examples are ECL [38] or GEM [39]. Bates emphasizes the benefits of using models for specifying expected behaviour [40], which is confirmed in a survey by Dwyer [41]. This direction is, among others, further developed in the work by Uchitel et al. on the specification on positive and negative scenarios [42] and Braberman et al. on the visual specification of timed event scenarios [43].

Event patterns as fragments of behaviour can also be specified using process model query languages, such as BP-QL [44] and BPMN-Q [45], [46]. BP-Mon [37] as discussed above, for instance, is directly grounded in BP-QL. Process model query languages are particularly suited for querying process-model-compliant behaviour that is enforced by a process engine. Our behavioural profiles-based approach permits the direct translation to event patterns typically supported by complex event processing engines. This means that work on querying languages can benefit from our optimisation rules once they are not used for querying the process specification, but instead monitoring event sequences at run-time.

More generally, our work relates to behavioural verification and compliance checking. Behavioural constraints as those encoded in behavioural profiles are investigated for compliance checking of causality [47], transition adjacency [48], [49] and for identifying root causes of non-compliance [50]. Recently, these concepts are adapted towards real-time monitoring of processes [51] and cross-organizational business processes [52]. These works can benefit from our optimisation rules that help to increase performance.

The assumption of a normative process model is related to transactional properties of business process implementations. Here, workflow transactions guarantee the adherence to specified behaviour. Various works adapt transaction models from databases (e.g., [53]) to workflow systems [54], [55]. These works can be seen as the foundations that enable the optimisation of event pattern matching using process models.

7 Conclusion

In this paper, we addressed the challenge of realizing complex event processing in an efficient manner. Under the assumption of a normative business process model, we showed how event pattern matching is optimised. We used behavioural constraints to rewrite event-based patterns, and select and transform execution plans. For the presented rules, we showed correctness and their potential for improving the processing, and discussed their completeness within the scope set. Applicability and effectiveness was demonstrated in a study with a large number of real-world processes from an insurance company. Also, we showed that our approach improves processing for sample applications with two state-of-the-art systems for stream processing and complex event detection, and that it connects well with existing optimisations for event pattern matching to achieve drastic performance improvements.

Our approach works well for expected events representing the accurate behaviour of the process or foreseen exceptional cases, and for accurate process models, where a technical process model is used for enactment. We foresee that process mining techniques [48] would alleviate some of these strict constraints by providing means to verify the accuracy of event streams with respect to an existing process models.

In future work, we intend to investigate further aspects of process models for optimisation. For instance, we may exploit constraints derived from data dependencies between activities.

References

[1] M. zur Muehlen and R. Shapiro, *Handbook on Business Process Management 2: Strategic Alignment, Governance, People and Culture*, Springer, 2010.

[2] M. B. Blake, D. J. Cummings, A. Bansal, and S. K. Bansal, "Workflow composition of service level agreements for web services," *DSS*, vol. 53, no. 1, pp. 234–244, 2012.

[3] A. Adi and O. Etzion, "Amit - the situation manager," *JVLDB*, vol. 13, no. 2, pp. 177–203, May 2004.

[4] M. Mendes, P. Bizarro, and P. Marques, "Benchmarking event processing systems: current state and future directions," in *Proc. First Joint WOSP/SIPEW Int. Conf. on Performance Engineering*, 2010, pp. 259–260.

[5] ebizQ, "Event processing market pulse," Tech. Rep., 2007. http://www.complexevents.com/2007/10/30/event-processing-market-pulse-2007/

[6] Y. Magid, G. Sharon, S. Arcushin, I. Ben-Harrush, and E. Rabinovich, "Industry experience with the ibm active middleware technology (amit) complex event processing engine," in *DEBS*, ACM, 2010, pp. 140–149.

[7] F. Leymann and D. Roller, *Production Workflow*, Prentice Hall, 2000.

[8] M. Weidlich, A. Polyvyanyy, J. Mendling, and M. Weske, "Causal behavioural profiles - efficient computation, applications, and evaluation," *Fundam. Inform.*, vol. 113, no. 3-4, pp. 399–435, 2011.

[9] M. Weidlich, J. Mendling, and M. Weske, "Efficient consistency measurement based on behavioral profiles of process models," *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 410–429, 2011.

[10] M. Weidlich, H. Ziekow, and J. Mendling, "Optimising complex event queries over business processes using behavioural profiles," in *BPM Workshops*, ser. LNBIP, vol. 66. Springer, 2010, pp. 743–754.

[11] M. Weske, *Business Process Management*. Springer-Verlag, 2007.

[12] Object Management Group (OMG), "Business Process Model and Notation (BPMN) Version 2.0," Tech. Rep., January 2011.

[13] N. Lohmann, E. Verbeek, and R. M. Dijkman, "Petri net transformations for business processes - a survey," *TOPNOC*, vol. 2, pp. 46–63, 2009.

[14] O. Etzion and P. Niblett, *Event Processing in Action*. Manning, 2010.

[15] EPCglobal, "EPC Information Services (EPCIS) V 1.01," http://www.epcglobalinc.org/standards/epcis/, 2007, last accessed Jan 27, 2012.

[16] G. Sharon and O. Etzion, "Event processing networks: model and implementation," *IBM System Journal*, vol. 47, no. 2, pp. 321–334, 2008.

[17] L. Brenna, J. Gehrke, M. Hong, and D. Johansen, "Distributed event stream processing with non-deterministic finite automata," in *DEBS*. ACM, 2009, pp. 1–12.

[18] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *SIGMOD*. ACM, 2006, pp. 407–418.

[19] M. Akdere, U. Çetintemel, and N. Tatbul, "Plan-based complex event detection across distributed sources," *VLDB*, vol. 1, no. 1, pp. 66–77, 2008.

[20] W. Aalst, *Business Process Management*. Springer, 2000, vol. LNCS 1806, ch. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques, pp. 161–183.

[21] M. Weidlich, H. Ziekow, A. Gal, J. Mendling, and M. Weske,"Supplemental Material: Optimising Event Pattern Matching using Business Process Models," http://matthiasweidlich.com/paper/cep_opt_supplement_TR_2013.pdf.

[22] W. V. der Aalst, K. V. Hee, J. V. der Werf, and M. Verdonk, "Auditing 2.0: Using process mining to support tomorrow's auditor," *IEEE Computer*, vol. 43, no. 3, pp. 90–93, 2010.

[23] J. Mendling, *Metrics for Process Models*, ser. LNBIP. Springer, 2008, vol. 6.

[24] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A general purpose event monitoring system," in *CIDR*. 2007, pp. 412–422.

[25] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing," in *CIDR*. 2007, pp. 363–374.

[26] F. Casati and A. Discenza, "Modeling and managing interactions among business processes," *JSI*, vol. 10, no. 2, pp. 145–168, 2001.

[27] G. Li, V. Muthusamy, and H.-A. Jacobsen, "A distributed service-oriented architecture for business process execution," *TWEB*, vol. 4, no. 1, 2010.

[28] J. J. King, "Quist: A system for semantic query optimization in relational databases," in *VLDB*. IEEE CS, 1981, pp. 510–517.

[29] U. S. Chakravarthy, J. Grant, and J. Minker, "Logic-based approach to semantic query optimization," *ACM TODS*, vol. 15, no. 2, pp. 162–207, 1990.

[30] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM TODS*, vol. 30, no. 1, pp. 122–173, 2005.

[31] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," in *SIGMOD*, 2002.

[32] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network stream query processing," in *PODS*. ACM, 2005.

[33] J. Shneidman, P. Pietzuch, M. Welsh, M. Seltzer, and M. Roussopoulos, "A cost-space approach to distributed query optimization in stream based overlays," in *ICDEW*. IEEE CS, 2005, p. 1182.

[34] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch, "Distributed complex event processing with query rewriting," in *DEBS*. ACM, 2009, pp. 1–12.

[35] E. Rabinovich, O. Etzion, and A. Gal, "Pattern rewriting framework for event processing optimization," in *DEBS*. ACM, 2011, pp. 101–112.

[36] L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W.-P. Hsiung, and K. S. Candan, "Runtime semantic query optimization for event stream processing," in *ICDE*. IEEE, 2008, pp. 676–685.

[37] C. Beeri, A. Eyal, T. Milo, and A. Pilberg, "Query-based monitoring of bpel business processes," in *SIGMOD*, ACM, 2007, pp. 1122–1124.

[38] C. Sánchez, S. Sankaranarayanan, H. Sipma, T. Zhang, D. L. Dill, and Z. Manna, "Event correlation: Language and semantics," in *EMSOFT*, ser. LNCS, vol. 2855. Springer, 2003, pp. 323–339.

[39] M. Mansouri-Samani and M. Sloman, "Gem: a generalized event monitoring language for distributed systems," *Distributed Systems Engineering*, vol. 4, no. 2, pp. 96–108, 1997.

[40] P. C. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," *ACM TCS*, vol. 13, no. 1, pp. 1–31, 1995.

[41] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE*. ACM, 1999, pp. 411–420.

[42] S. Uchitel, J. Kramer, and J. Magee, "Incremental elaboration of scenario-based specifications and behavior models using implied scenarios," *ACM TOSEM*, vol. 13, no. 1, pp. 37–85, 2004.

[43] V. A. Braberman, N. Kicillof, and A. Olivero, "A scenario-matching approach to the description and model checking of real-time properties," *IEEE TSE*, vol. 31, no. 12, pp. 1028–1041, 2005.

[44] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo, "Querying business processes with bp-ql," *Inf. Syst.*, vol. 33, no. 6, pp. 477–507, 2008.

[45] A. Awad, G. Decker, and M. Weske, "Efficient compliance checking using BPMN-Q and temporal logic," in *BPM*, ser. LNCS, vol. 5240. Springer, 2008, pp. 326–341.

[46] A. Awad and S. Sakr, "On efficient processing of BPMN-Q queries," *Computers in Industry*, vol. 63, no. 9, pp. 867–881, 2012.

[47] J. Hoffmann, I. Weber, and G. Governatori, "On compliance checking for clausal constraints in annotated process models," *Information Systems Frontiers*, vol. 14, no. 2, pp. 155–177, 2012.

[48] W. M. P. van der Aalst, *Process Mining*. Springer, 2011.

[49] H. Zha, J. Wang, L. Wen, C. Wang, and J. Sun, "A workflow net similarity measure based on transition adjacency relations," *Computers in Industry*, vol. 61, no. 5, pp. 463–471, 2010.

[50] M. Weidlich, A. Polyvyanyy, N. Desai, J. Mendling, and M. Weske, "Process compliance analysis based on behavioural profiles," *Inf. Syst.*, vol. 36, no. 7, pp. 1009–1025, 2011.

[51] W. M. P. van der Aalst, K. M. van Hee, J. M. E. M. van der Werf, A. Kumar, and M. Verdonk, "Conceptual model for online auditing," *DSS*, vol. 50, no. 3, pp. 636–647, 2011.

[52] B. Wetzstein, D. Karastoyanova, O. Kopp, F. Leymann, and D. Zwink, "Cross-organizational process monitoring based on service choreographies," in *SAC*. ACM, 2010, pp. 2485–2490.

[53] H. Garcia-Molina and K. Salem, "Sagas," *ACM SIGMOD Record*, vol. 16, no. 3, pp. 249–259, 1987.

[54] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth, "An overview of workflow management: From process modeling to workflow automation infrastructure," *DPD*, vol. 3, no. 2, pp. 119–153, 1995.

[55] G. Alonso, D. Agrawal, A. E. Abbadi, M. Kamath, R. Günthör, and C. Mohan, "Advanced transaction models in workflow contexts," in *ICDE*. IEEE CS, 1996, pp. 574–581.

Matthias Weidlich is a research associate at Imperial College London. Before, he was a research fellow at the Technion - Israel Institute of Technology. He received his PhD from the Hasso Plattner Institute (HPI), University of Potsdam, Germany, in 2011. His research focuses on formal methods for processes, event processing, and data integration. His results appeared in journals, such as IEEE Transactions on Software Engineering, Information Systems, and The Computer Journal. He is on the editorial board of Elsevier's Information Systems and a member of ACM and IEEE.

Holger Ziekow has been a senior researcher and project manager at AGT International since 2011. In 2010 he worked as a postdoctoral researcher at the Humboldt-University Berlin and in 2009 at the International Computer Science Institute in Berkeley, California. He holds a Diplom from Technical University of Berlin and a Ph.D. from the Humboldt-University Berlin.

Avigdor Gal is an Associate professor at the Faculty of Industrial Engineering & Management at the Technion - Israel Institute of Technology. He received his D.Sc. degree from the Technion in 1995 in the area of temporal active databases. He has published more than 100 papers in journals, books and conferences on event processing, business process management, and data integration. Avigdor is the author of the book Uncertain Schema Matching. He served as a Program co-Chair and General Chair of DEBS, BPM, and CoopIS. He is an Area Editor of the Encyclopedia of Database Systems.

Jan Mendling is a Full Professor at the Institute for Information Business, WU Vienna. His research areas include Business Process Management, Conceptual Modelling and Enterprise Systems. He received a PhD degree from WU Vienna and diploma degrees from University of Trier. He also worked with QUT Brisbane and HU Berlin. Jan has published more than 200 research papers and articles. He is member of the editorial board of three international journals, one of the founders of the Berliner BPM-Offensive, and board member of the Austrian Gesellschaft für Prozessmanagement.

Mathias Weske is a Full Professor and chair of the business process technology research group at Hasso Plattner Institute at the University of Potsdam. His research interests include all aspects of business process management. Dr. Weske is on the steering committee of the BPM conference series, and he leads the BPM Academic Initiative.