Scenario-Based Process Querying for Compliance, Reuse, and Standardization

Artem Polyvyanyy[a,*], Anastasiia Pika[b], Arthur H. M. ter Hofstede[b]

[a]*The University of Melbourne, Parkville, VIC 3010, Australia*
[b]*Queensland University of Technology, 2 George St., Brisbane, QLD 4000, Australia*

Abstract

Process models constitute valuable artifacts for organizations. A process model formally captures the way an organization works internally and interacts with its customers and partners. Over time, more models may be created as business practices evolve (leading to different versions of models) or an organization expands, e.g., through mergers or acquisitions. It is not uncommon for large organizations to have to manage thousands of process models. Retrieval of process models with desired properties then poses a significant challenge, particularly when one is concerned with finding models that describe certain process scenarios, i.e., sequences of tasks captured in models. This paper proposes a method for automated process model retrieval based on scenario compatibility. A process model is retrieved if it has the potential to perform the specified process scenario. To allow for scenarios to be underspecified, wildcards may be used in their description. The paper reports on a formal language for scenario-based process querying, its implementation, and evaluation in the context of industrial and synthetic process models. The results show that the technique works in (close to) real time.

Keywords: Process querying, process scenario, process instance, process querying method, process query language, PQL

1. Introduction

To remain competitive, organizations continuously streamline their (business) processes in response to internal and external changes and challenges. Business Process Management (BPM) [1, 2] provides a systematic approach to analyzing and improving processes in organizations. Formal representations of processes, so-called *process models*, act as key artifacts in analysis and improvement of processes. Given the time and effort it may take to create process models, they constitute a significant investment by the organization. The key focus of process models is in activities/tasks involved in the conduct of processes and the order in which the tasks need to be performed. As such, a process model describes the *scenarios* that can play out during its execution.

*Corresponding author
Email addresses: artem.polyvyanyy@unimelb.edu.au (Artem Polyvyanyy), a.pika@qut.edu.au (Anastasiia Pika), a.terhofstede@qut.edu.au (Arthur H. M. ter Hofstede)

Over time, organizations may accumulate hundreds [3, 4] or even thousands [5, 6] of process models due to an increased process focus, changes to business practices, mergers, or acquisitions. In this context, it is challenging to quickly and accurately *retrieve* (filter out) models that describe certain desired or undesired scenarios; e.g., to support the update of existing models or the creation of new models that are based on existing models. Retrieval of models should be based on the scenarios that models specify (semantics) and not on the particular form models may take (syntax), as stakeholders tend to understand scenarios and not the way these scenarios are encoded in models.

Scenarios are widely used when modeling components of distributed systems, telecommunication systems, and business processes [7, 8, 9, 10, 11, 12, 13], as scenarios remind storyboards that are generally accepted to be easier to comprehend than process models [14, 15]. The individual scenarios are then employed to discover or automatically synthesize complex process models. In this paper, we use process scenario templates to trigger retrieval of process models that describe scenarios that match the template. A solution to this problem can support process compliance [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27], reuse [28, 29], and standardization [30, 31] use cases in BPM [32].

Process querying studies automated methods for managing, e.g., retrieving or manipulating, repositories of models that describe observed and/or envisioned processes, and relationships between these processes [33]. A process querying method is a technique that given a process repository and a formal specification of an instruction to manage the repository, i.e., a *process query*, systematically implements the query in the repository. A number of languages for specifying process queries have been developed over time, though most of these are (primarily) based on model structure and not on the scenarios models encode [33, 34]. A notable exception is the Process Query Language (PQL) [35, 36]. PQL is an SQL-like language for managing process repositories that is grounded in the semantics of process models. It aims at implementing the *process querying compromise*, refer to Section 4.4 in [33], by supporting *useful* and *efficiently computable* process queries. The approach for scenario-based retrieval of process models proposed in this paper is implemented as an extension of PQL.

Concretely, this paper contributes:

- Formalization of the scenario-based process querying problem for retrieving process models based on the scenarios that they describe;
- A solution to the scenario-based process querying problem that justifies that it is indeed *computable*, including proofs and discussions of correctness and complexity of the proposed solution. Note that the *model checking* problem [37] for process models, which is a generalization of the problem addressed in this paper, is *undecidable* [38]; we refer the reader to Section 8 for details.
- A publicly available implementation of the scenario-based process querying problem as an extension to PQL[1], which includes extensions to its abstract syntax, concrete syntax, and dynamic semantics;
- A quantitative evaluation of our implementation of the scenario-based process querying problem that demonstrates the feasibility of using the approach in industrial

[1]The implementation is available at: `https://github.com/processquerying/PQL`.

settings in (close to) *real time*.

The rest of the paper proceeds as follows. The next section motivates the need for scenario-based process querying to support the BPM use cases of compliance, reuse, and standardization, elicits requirements from these use cases, and presents motivating examples. Section 3 introduces preliminary notions used to support the subsequent discussions. Section 4 gives a rigorous definition of the scenario-based process querying problem and method that addresses the requirements identified in Section 2. An extension of PQL for supporting scenario-based process querying is presented in Section 5. Section 6 gives a solution to the scenario-based process querying problem. Section 7 presents the results of a quantitative evaluation of our implementation of the querying method that justifies the feasibility of using the method in industrial settings. Section 8 discusses the expressiveness, computability, and complexity of the method, as well as gives a comprehensive comparison with a related area of model checking. Section 9 provides an overview of related work. Finally, Section 10 summarizes the paper and discusses avenues for future work.

2. Background

The aim of this section is threefold: Section 2.1 motivates the need for scenario-based process querying to support the BPM use cases of process compliance, reuse, and standardization. Section 2.2 draws three requirements from these use cases. Finally, Section 2.3 presents motivating examples that address the identified requirements.

2.1. Process Compliance, Reuse, and Standardization

Process compliance, reuse, and standardization are three actively researched areas in BPM [32]. Next, we briefly introduce these areas and discuss how they can benefit from the use of scenario-based process querying.

Process models are subject to constraints enforced by regulations and/or laws, often referred to as compliance rules [39, 40]. Verification of regulatory *process compliance* is usually formulated as a retrieval of process models that describe certain non-compliant scenarios [41]. For example, compliance rule CR1 from [39] states: "Packages Known to be Held by a Regulatory Authority must not be Routed by a Sort Officer until the Package is Known to be Cleared by the Regulatory Authority". Effective compliance checking requires the retrieval of information about process instances.

Process reuse refers to the problem of constructing new process models by assembling already designed ones [42]. To this end, existing process models, fragments, and patterns are taken from other contexts instead of creating them from scratch [43]. When developing new or modifying existing process models, one can reuse information that is contained in other process models [44, 45, 46, 47, 48, 49], for example those models that describe the process scenarios of interest. Recent studies report that process reuse is done mostly manually, both by practitioners and academics, due to the absence of advance search capabilities [42].

Standard process models are exemplar models that should be used as references [50]. *Process standardization* refers to the act of replacing different but similar process models, or model fragments, with a single unified model, or fragment [51]. The standardized,

also referred to as harmonized or consolidated, process models/fragments encode best practices for handling similar process scenarios [30]. Scenario-based process querying can be used at early stages of business process standardization initiatives to identify process models/fragments that describe similar scenarios.

2.2. Requirements Analysis

To elicit functional requirements for scenario-based process querying, we conducted a systematic literature review. The review was accomplished according to the guidelines in [52]. The scope of the review was concerned with the use cases of process compliance, reuse, and standardization. The literature search was performed using Scopus, which is a well-known database of peer-reviewed literature and citations. Three searches were performed (one for each use case), all for the literature in the subject area of *computer science* and *article* document types. The literature was retrieved based on the presence of these keywords in abstracts, keywords, and document titles (per use case):

- "process compliance";
- "process reuse"; and
- "process standardization", "process standardisation".

These searches discovered 116 distinct articles.[2] We could not retrieve the full content of 11 articles and 1 article was not in English. After close examination of the full content of the remaining 104 articles, 41 articles were identified as relevant; these articles address the use cases of process compliance (27 articles), reuse (7), and standardization (7), as those were introduced in Section 2.1. Examples of areas of irrelevant articles include social networks, security, product design, etc.

Out of 41 relevant articles, 18 discuss the use of scenarios for retrieving process models. Based on the detailed analysis of these 18 articles, we identified three functional requirements for querying repositories of process models:

R1 *Exact scenario matching*. To retrieve process models that describe a given scenario provided as a sequence of performed tasks;

R2 *Partial scenario matching*. To retrieve process models that describe a given partially specified scenario provided as a sequence of performed tasks that allows the presence of not specified tasks at some given positions in the sequence;

R3 *Task label similarity*. To retrieve process models that describe a given (partial) scenario using task labels that are similar to those used in the given scenario.

Table 1 summarizes the origins of the requirements in the retrieved literature.

Table 1: Literature that supports the three requirements for scenario-based process querying.

Use case	R1	R2	R3
Compliance	[26, 53, 22]	[16, 17, 19, 20, 21, 24, 25]	[26, 18, 22, 23]
Reuse		[28, 29]	[28, 29]
Standardization		[30, 31]	[31]

[2]The search and all the corresponding analysis results were updated on May 22, 2019.

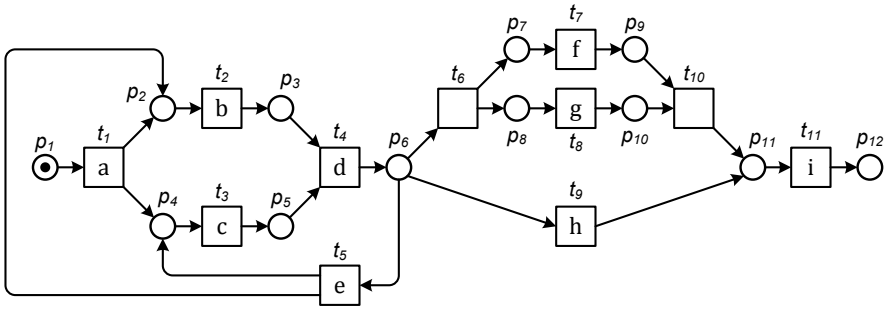


Figure 1: A net system, where a := "receive travel request", b := "book flight", c := "book hotel", d := "verify travel booking", e := "reject booking", f := "book travel", g := "archive successful booking request", h := "archive unsuccessful booking request", and i := "finalize booking".

2.3. Motivating Examples

This section presents motivating examples of scenario-based process querying that demonstrate the requirements from Section 2.2. Note that all the subsequently proposed examples address the retrieval of the process model in Figure 1.[3]

R1 *Exact scenario matching*. The process model in Figure 1 should be retrieved as the result of process querying based on the scenario `<a,b,c,d,h,i>`, as it describes the scenario that starts by executing task `a`, followed by tasks `b`, `c`, `d`, and `h` (in the proposed order), and concludes by executing task `i`, without executing any other task; the proposed short task names are specified in the caption of Figure 1.

R2 *Partial scenario matching*. The process model in Figure 1 should be retrieved as the result of process querying based on the partial scenario specification that says that a scenario of interest starts with task `a`, which is eventually followed by task `d`, immediately followed by task `e`, eventually followed by task `g`, immediately followed by task `f`, followed by some other tasks. We capture such a partial scenario using the template `<a,*,d,e,*,g,f,*>`, where the special asterisk symbol `'*'` stands for an arbitrary sequence of tasks. The model should be retrieved because, among other matching scenarios, it describes the matching scenario `<a,b,c,d,e,b,c,d,g,f,i>`.

R3 *Task label similarity*. A user of scenario-based process querying may be unaware of the exact labels used to specify tasks in process models. Hence, she should be able to retrieve models with tasks that are similar to those used in a query. Given that task `j` := "archive booking request" is accepted as sufficiently similar to task `g`, the model in Figure 1 should be retrieved as the result of process querying based on the scenario template `<a,*,d,e,*,~j,f,*>`, where task `j` with the special tilde grapheme `'~'` in front refers to any task that is similar to `j`.

Referring to CR1 from Section 2.1, process models that violate CR1 are those that match the scenario template `<*,` ~"Hold Package", `*,` ~"Route Package", `*,` ~"Clear Package", `*>`; note that a subsequent check of resource roles is required.

[3]The process model in Figure 1 is a (Petri) net system; this formalism is explained in Section 3.2.

3. Preliminaries

This section recalls basic concepts and notations that are related to *Petri nets* and *alignments* between traces and executions of Petri nets. These concepts will be used to support later discussions.

3.1. Multisets, Sequences, Languages, and Functions

A *multiset*, or a *bag*, is a generalization of the concept of a set that allows a multiset to contain multiple instances of the same element. Multisets can be used to encode states of Petri nets. Moreover, an event log is usually formalized as a multiset of traces. By $\mathcal{B}(A)$, we denote the set of all finite multisets over some set A. For some multiset $B \in \mathcal{B}(A)$, $B(a)$ denotes the multiplicity of element a in B, i.e., the number of times element $a \in A$ appears in B. For example, $B_1 := [\,]$, $B_2 := [a,b,b]$, and $B_3 := [b^2,a]$ are multisets over the set $\{a,b\}$. For the above multisets, it holds that B_1 is *empty*, $B_2(a) = 1 = B_3(a)$, $B_2(b) = 2 = B_3(b)$, and $B_2 = B_3$.

The standard set operations have been extended to deal with multisets as follows. If element a is a member of multiset B, this is denoted by $a \in B$, while if element b is not a member of B, we write $b \notin B$. The union of two multisets C and D, denoted by $C \uplus D$, is the multiset that contains all elements of C and D such that the multiplicity of an element in the resulting multiset is equal to the sum of multiplicities of this element in C and D. The difference of two multisets C and D, denoted by $C \setminus D$, is the multiset that for each element $x \in C$ contains $max(0, C(x) - D(x))$ occurrences of x.

In mathematics, a *sequence* is an ordered list of elements. We use sequences to capture traces in event logs and orderings of transition occurrences in Petri nets. By $\sigma := \langle a_1, a_2, \ldots, a_n \rangle \in A^*$, we denote a sequence over some set A of length $n \in \mathbb{N}_0$, $a_i \in A$, $i \in [1..n]$.[4] By $\sigma_{[i]}$, $i \in [1..n]$, we refer to the i-th element of σ, i.e., $\sigma_{[i]} = a_i$. Given a sequence σ and a set K, by $\sigma|_K$, we denote a sequence obtained from σ by deleting all elements of σ that are not members of K without changing the order of the remaining elements. Given two sequences σ and σ', by $\sigma \circ \sigma'$, we denote the *concatenation* of σ and σ', i.e., the sequence obtained by appending σ' to the end of σ. For example, $\langle \mathtt{a},\mathtt{b},\mathtt{a} \rangle \circ \langle\rangle \circ \langle \mathtt{b},\mathtt{a} \rangle = \langle \mathtt{a},\mathtt{b},\mathtt{a},\mathtt{b},\mathtt{a} \rangle$; $\langle\rangle$ is the empty sequence. For two sets of sequences S_1 and S_2 over A, $S_1 \circ S_2 := \{\sigma \in A^* \mid \exists\, \sigma_1 \in S_1 \exists\, \sigma_2 \in S_2 : \sigma = \sigma_1 \circ \sigma_2\}$. By $\mathit{suffix}(\sigma, i)$, $i \in \mathbb{N}$, we denote the suffix of σ starting from and including position i. For example, let $\sigma := \langle \mathtt{a},\mathtt{b},\mathtt{a},\mathtt{b},\mathtt{a},\mathtt{h},\mathtt{a},\mathtt{l},\mathtt{a},\mathtt{m},\mathtt{a},\mathtt{h},\mathtt{a} \rangle$ be a sequence. Then, it holds that $\mathit{suffix}(\sigma, 6) = \langle \mathtt{h},\mathtt{a},\mathtt{l},\mathtt{a},\mathtt{m},\mathtt{a},\mathtt{h},\mathtt{a} \rangle$.

An *alphabet* is any nonempty finite set. The elements of an alphabet are its *symbols*. These are two example alphabets:

$$\Sigma_1 := \{\mathtt{0},\mathtt{1},\mathtt{2},\mathtt{3},\mathtt{4},\mathtt{5},\mathtt{6},\mathtt{7},\mathtt{8},\mathtt{9},\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e},\mathtt{f}\}$$
$$\Sigma_2 := \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{g},\mathtt{h},\mathtt{i},\mathtt{j},\mathtt{k},\mathtt{l},\mathtt{m},\mathtt{n},\mathtt{o},\mathtt{p},\mathtt{q},\mathtt{r},\mathtt{s},\mathtt{t},\mathtt{u},\mathtt{v},\mathtt{w},\mathtt{x},\mathtt{y},\mathtt{z}\}$$

A *string* over an alphabet is a finite sequence of symbols from the alphabet. The symbols of a string are usually written next to one another, e.g., `7e0` and `halamaha`, are strings

[4] $\mathbb{N}_0$ denotes the set of all natural numbers including zero.

over Σ_1 and Σ_2, respectively. The string of length zero is called the *empty string* and is denoted by ε. Finally, a (formal) *language* over an alphabet Σ is a set of strings over Σ.

Let $k := (k_1, k_2, \ldots, k_n) \in K_1 \times K_2 \times \ldots \times K_n$ be a point in n-dimensional space, where $K_1, K_2, \ldots, K_n$ are some sets. The *projection function* $\pi_i(k)$, $i \in [1..n]$, is defined as $\pi_i(k) := k_i$, where k_i is the i-th coordinate of k. Let $\kappa := \langle \kappa_1, \kappa_2, \ldots, \kappa_m \rangle$, $m \in \mathbb{N}_0$, where $\kappa_j \in K_1 \times K_2 \times \ldots \times K_n$, $j \in [1..m]$, be a sequence of points in n-dimensional space. Then, $\pi_i(\kappa) := \langle \pi_i(\kappa_1), \pi_i(\kappa_2), \ldots, \pi_i(\kappa_m) \rangle$, $i \in [1..n]$. Finally, if $\sigma := \langle a_1, a_2, \ldots, a_n \rangle \in A^*$ is a sequence over A and f is a function over A, then $f(\sigma) := \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$. Similarly, if $A' \subseteq A$, then $f(A') := \{f(a) \mid a \in A'\}$.

3.2. Petri Nets and Net Systems

A Petri net is a model of a distributed system [54]. Let $\mathbb{A}$ be a universe of labels.

Definition 3.1 (Petri net)**.**
A *(labeled) Petri net*, or a *net*, is a 5-tuple $N := (P, T, F, \Lambda, \lambda)$, where P is a finite set of *places*, T is a finite set of *transitions*, $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, $\Lambda \subset \mathbb{A}$ is a set of *labels*, such that P, T, and $\mathbb{A}$ are pairwise disjoint, and $\lambda : T \to \Lambda \cup \{\tau\}$ is a function that relates each transition to its label, where τ is a special label, $\tau \notin P \cup T \cup \mathbb{A}$. ⌟

Places and transitions are conjointly referred to as *nodes* of the net. A node $x \in P \cup T$ is an *input* node of a node $y \in P \cup T$ iff $(x, y) \in F$. Similarly, a node $x \in P \cup T$ is an *output* node of a node $y \in P \cup T$ iff $(y, x) \in F$. By $\bullet x$, $x \in P \cup T$, we denote the *preset* of x, i.e., the set of all input nodes of x, while by $x\bullet$, we denote the *postset* of x, i.e., the set of all output nodes of x. For a set of nodes $X \subseteq P \cup T$, $\bullet X := \bigcup_{x \in X} \bullet x$ and $X\bullet := \bigcup_{x \in X} x\bullet$.

Let $N := (P, T, F, \Lambda, \lambda)$ be a net. If $\lambda(t) = \tau$, $t \in T$, then t is *silent*; otherwise t is *observable*. Observable transitions are used to represent activities from the problem domain, whereas silent are used to encode systems' internal actions.

The execution semantics of Petri nets is defined in terms of states and state transitions. A state of a Petri net is captured by the concept of a *marking*.

Definition 3.2 (Marking)**.**
A *marking* of a net $N := (P, T, F, \Lambda, \lambda)$ is a multiset over its places $M \in \mathcal{B}(P)$. ⌟

A marking M of a Petri net $N := (P, T, F, \Lambda, \lambda)$ is often interpreted as an assignment of *tokens* to places, i.e., marking M 'puts' $M(p)$ tokens at place p, $p \in P$. A net system is a Petri net with an initial marking and a final marking.

Definition 3.3 (Net system)**.**
A *net system*, or a *system*, is a 3-tuple $S := (N, M_{ini}, M_{fin})$, where $N := (P, T, F, \Lambda, \lambda)$ is a net, $M_{ini} \in \mathcal{B}(P)$ is the *initial* marking of N, and $M_{fin} \in \mathcal{B}(P)$ is the *final* marking of N. ⌟

By $\mathbb{S}$, we denote the universe of net systems. Net systems have a well-established graphical notation. In this notation, places are visualized as circles, transitions are drawn as rectangles, every pair of nodes (x, y) in the flow relation is depicted as a directed arc that leads from x to y, and tokens induced by the initial marking are depicted as black dots inside the assigned places. For each observable transition, its label is depicted inside the corresponding rectangle; silent transitions are drawn as empty rectangles. Figure 1 visualizes a net system (N, M_{ini}, M_{fin}), $N := (P, T, F, \Lambda, \lambda)$, where $P := \{p_1, \ldots, p_{12}\}$,

$T := \{t_1, \ldots, t_{11}\}$, F is defined by the directed edges of the graph depicted in Figure 1, $\Lambda := \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}\}$, $\lambda := \{(t_1, \mathtt{a}), (t_2, \mathtt{b}), (t_3, \mathtt{c}), (t_4, \mathtt{d}), (t_5, \mathtt{e}), (t_6, \tau), (t_7, \mathtt{f}), (t_8, \mathtt{g}), (t_9, \mathtt{h}), (t_{10}, \tau), (t_{11}, \mathtt{i})\}$, $M_{ini} := [p_1]$, and $M_{fin} := [p_{12}]$. Note the use of short labels in the formalism and Figure 1; refer to the caption of Figure 1 for the complete label names. Also note that there is no explicit visual notation used to encode the final marking of the net.

Let $N := (P, T, F, \Lambda, \lambda)$ be a net. A transition $t \in T$ is *enabled* in a marking M of N, denoted by $(N, M)[t\rangle$, iff every input place of t contains at least one token, i.e., $\forall p \in \bullet t : M(p) > 0$. If a transition $t \in T$ is enabled in a marking M of N, then t can *occur*, which *leads* to a fresh marking $M' := (M \setminus \bullet t) \uplus t\bullet$ of N, where $\uplus$ denotes the multiset union, i.e., transition t 'consumes' one token from every input place of t and 'produces' one token for every output place of t. By $(N, M)[t\rangle(N, M')$, we denote the fact that an occurrence of t leads from M to M'. For example, it holds that $(N, M_{ini})[t_1\rangle(N, [p_2, p_4])$, where N and M_{ini} are the net and the initial marking of the net system in Figure 1.

A finite sequence of transitions $\sigma := \langle t_1, t_2, \ldots, t_n\rangle \in T^*$, $n \in \mathbb{N}_0$, is an *occurrence sequence* of a net system $S := (N, M_{ini}, M_{fin}) \in \mathbb{S}$, $N := (P, T, F, \Lambda, \lambda)$, iff σ is empty or there exists a sequence of markings $\langle M_0, M_1, \ldots, M_n\rangle$, such that $M_0 = M_{ini}$ and for every position $i \in [1..n]$ in σ it holds that $(N, M_{i-1})[t_i\rangle(N, M_i)$; we say that σ *leads* from M_0 to M_n and denote this fact by $(N, M_0)[\sigma\rangle(N, M_n)$. An occurrence sequence σ of S is an *execution* iff σ leads from M_{ini} to M_{fin}. For example, two sequences of transitions $\langle t_1, t_2, t_3, t_4, t_5\rangle$ and $\langle t_1, t_3, t_2, t_4, t_9, t_{11}\rangle$ are two occurrence sequences of the net system in Figure 1, whereas the latter is also an execution. By $\mathbb{E}_S$, we denote the set of all executions of S.

A net $N := (P, T, F, \Lambda, \lambda)$ is a *workflow net* iff N has a dedicated initial place $i \in P$, a dedicated final place $f \in P$, and every node of N is on a directed path from i to f [55]. A net system $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, is a *workflow system* iff N is a workflow net with a dedicated initial place $i \in P$, a dedicated final place $f \in P$, and it holds that $M_{ini} = [i]$ and $M_{fin} = [f]$. Note that the net system in Figure 1 is a workflow system with the initial place p_1 and the final place p_{12}.

3.3. Traces and Optimal Alignments

A *trace* captures one execution of a dynamic system, i.e., a *process scenario*.

Definition 3.4 (Trace)**.**
A *trace* $\upsilon \in \mathbb{A}^*$ is a finite sequence of labels. ⌟

Every label of a trace represents an *event*, i.e., an occurrence of an activity of a system. The order in which labels appear in a given trace encodes the order in which the corresponding activities were observed/recorded. For instance, $\upsilon_1 := \langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{g}, \mathtt{f}, \mathtt{i}\rangle$ and $\upsilon_2 := \langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}, \mathtt{c}, \mathtt{b}, \mathtt{e}, \mathtt{h}, \mathtt{i}\rangle$ are two example traces.

In a perfect world, every trace υ of a dynamic system S, e.g., a net system, describes some execution of S, i.e., υ fits S perfectly. Formally, a trace υ *fits perfectly* a net system $S \in \mathbb{S}$ if and only if there exists an execution $\sigma \in \mathbb{E}_S$ for which it holds that $\upsilon = \lambda(\sigma)|_{\mathbb{A}}$. In the real world, however, recorded traces can deviate from executions prescribed by models. Given a trace υ and a model of a system S, *conformance checking* [56] studies

whether υ is in accordance with some execution of S. One can measure the conformance of traces and net systems using alignments [57]. An *alignment* between a trace and an execution of a net system can be formalized as a sequence of legal moves, where a *move* is a pair in which the first component refers to a label in the trace and the second component refers to a transition in the execution.

Definition 3.5 (Move, Legal move)**.**
A *move* over a net system $S := (N, M_{ini}, M_{fin}) \in \mathbb{S}$, where $N := (P, T, F, \Lambda, \lambda)$, is a pair $(x, y) \in (\mathbb{\Lambda} \cup \{\gg\}) \times (T \cup \{\gg\}) \setminus \{(\gg, \gg)\}$, where $\gg \notin P \cup T \cup \mathbb{\Lambda} \cup \{\tau\}$ is a special "*no move*" element.
- Move (x, y) is a *move on trace* iff $x \in \mathbb{\Lambda}$ and $y = \gg$.
- Move (x, y) is a *move on system* iff $x = \gg$ and $y \in T$.
- Move (x, y) is a *synchronous move* iff $x \in \Lambda$, $y \in T$, and $\lambda(y) = x$.

Move (x, y) is a *legal move* over S if it is either a move on trace, a move on system, or a synchronous move; otherwise move (x, y) is an *illegal move* over S. ⌟

By $\mathbb{M}_S$, we denote the set of all legal moves over a net system $S \in \mathbb{S}$. Finally, the notion of an alignment is specified below.

Definition 3.6 (Alignment)**.**
An *alignment* between a trace $\upsilon \in \mathbb{\Lambda}^*$ and an execution σ of a net system $S := (N, M_{ini}, M_{fin}) \in \mathbb{S}$, where $N := (P, T, F, \Lambda, \lambda)$, is a finite sequence $\gamma \in \mathbb{M}_S^*$ of legal moves over S for which it holds that $\pi_1(\gamma)|_{\mathbb{\Lambda}} = \upsilon$ and $\pi_2(\gamma)|_T = \sigma$. ⌟

Given a trace, one can compare two alignments between this trace and two different executions of a net system by associating costs with moves of these alignments. A *cost function on legal moves* over a net system $S \in \mathbb{S}$ is a function $c : \mathbb{M}_S \to \mathbb{N}_0$ that assigns costs to legal moves over S. The *cost of an alignment* γ between a trace $\upsilon \in \mathbb{\Lambda}^*$ and an execution σ of a net system $S \in \mathbb{S}$ as per a cost function c on legal moves over S is denoted by $c(\gamma)$ and is the sum of costs of all moves of γ, i.e., $c(\gamma) := \sum_{i=1}^{|\gamma|} c(\gamma_{[i]})$.

Given a trace $\upsilon \in \mathcal{A}^*$ and a net system $S := (N, M_{ini}, M_{fin}) \in \mathbb{S}$, $N := (P, T, F, \Lambda, \lambda)$, by $\mathbb{A}_S^\upsilon$ we denote the set that for every execution $\sigma \in \mathbb{E}_S$ contains all alignments between υ and σ, i.e., $\mathbb{A}_S^\upsilon := \{\gamma \in \mathbb{M}_S^* \mid \exists\, \sigma \in \mathbb{E}_S : \pi_1(\gamma)|_{\mathbb{\Lambda}} = \upsilon \wedge \pi_2(\gamma)|_T = \sigma\}$. The "cheapest" alignments between a trace ν and executions of a system S are *optimal alignments* between ν and S.

Definition 3.7 (Optimal alignment)**.**
An *optimal alignment* between a trace $\upsilon \in \mathbb{\Lambda}^*$ and a system $S \in \mathbb{S}$ as per a cost function c on legal moves over S is an alignment $\gamma \in \mathbb{A}_S^\upsilon$ such that $\forall \gamma' \in \mathbb{A}_S^\upsilon : c(\gamma) \leq c(\gamma')$. ⌟

The reader can refer to [58] for techniques to compute optimal alignments between traces and executions of systems. These techniques work on input net systems that describe at least one execution. Consequently, we adopt the same requirement and expect that for every system $S \in \mathbb{S}$ it holds that $\mathbb{E}_S \neq \varnothing$; for example $\mathbb{S}$ can correspond to the class of all *weak-sound* workflow systems [59], often also referred to as *easy-sound* workflow systems [58].

An optimal alignment between a trace and execution of a net system encodes one (out of possibly many) best effort fit between the trace and execution as per the

$\gamma_1^1 :=$

a	b	c	d	≫	g	f	≫	i
a	b	c	d	τ	g	f	τ	i
t_1	t_2	t_3	t_4	t_6	t_8	t_7	t_{10}	t_{11}

$\gamma_2^1 :=$

a	b	c	e	≫	c	b	e	h	i
a	b	c	≫	d	≫	≫	≫	h	i
t_1	t_2	t_3		t_4				t_9	t_{11}

$\gamma_2^2 :=$

a	b	c	≫	e	c	b	e	≫	h	i
a	b	c	d	e	c	b	≫	d	h	i
t_1	t_2	t_3	t_4	t_5	t_3	t_2		t_4	t_9	t_{11}

$\gamma_2^3 :=$

a	b	c	≫	e	c	b	≫	e	h	i
a	b	c	d	e	c	b	d	≫	h	i
t_1	t_2	t_3	t_4	t_5	t_3	t_2	t_4		t_9	t_{11}

Figure 2: Alignments between sample traces and the system in Figure 1.

employed cost function. Figure 2 shows four alignments between example traces υ_1 and υ_2 proposed earlier in this section and executions of the net system in Figure 1.

An alignment is visualized as a table in which moves are encoded as columns, where two successive columns refer to two successive moves. Each column of such a table has three rows. The top row contains the first component of the move, which is either an event of the trace or the special "no move" element '≫'. The bottom row is kept empty if the second component of the move is the special "no move" element; otherwise, it contains the second component of the move, i.e., a transition. The middle row contains '≫' if the second component of the move is the special "no move" element; otherwise, it contains the label assigned to the transition in the second component of the move. For example, alignment γ_1^1 from Figure 2 defines the sequence of legal moves $\langle(\mathtt{a},t_1),(\mathtt{b},t_2),(\mathtt{c},t_3),(\mathtt{d},t_4),(\gg,t_6),(\mathtt{g},t_8),(\mathtt{f},t_7),(\gg,t_{10}),(\mathtt{i},t_{11})\rangle$. Alignment γ_1^1 is an alignment between trace $\upsilon_1 := \langle\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{g},\mathtt{f},\mathtt{i}\rangle$ and execution $\langle t_1,t_2,t_3,t_4,t_6,t_8,t_7,t_{10},t_{11}\rangle$ of the net system S in Figure 1. Using γ_1^1, it is easy to see that υ_1 fits S perfectly. This is not the case for trace $\upsilon_2 := \langle\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{e},\mathtt{c},\mathtt{b},\mathtt{e},\mathtt{h},\mathtt{i}\rangle$. Alignments γ_2^1, γ_2^2, and γ_2^3, are three alignments between υ_2 and two different executions of S. Given the cost function c on legal moves over S that assigns the cost of one to all moves on trace and to all moves on system for which the second component is an observable transition, and gives the cost of zero to all other moves in $\mathbb{M}_S$, it holds that γ_2^2 and γ_2^3 are optimal alignments between υ_2 and S as per c, such that $c(\gamma_2^2) = 3 = c(\gamma_2^3)$; note that $c(\gamma_2^1) = 5$. In the literature, such a cost function c is called the *standard cost function* [58]. Given a trace υ, a net system S, and a cost function c on legal moves over S, $\mathbb{O}_{S,c}^{\upsilon}$ denotes the set of all optimal alignments between υ and S as per c. Note that $c(\gamma_1^1) = 0$ and thus, trivially, $\gamma_1^1 \in \mathbb{O}_{S,c}^{\upsilon_1}$.

4. Scenario-Based Process Querying

A *process querying method* is a technique that given a repository of process models and a formal instruction to manage the repository implements the instruction in the repository [33]. In this section, we propose a method for managing (a repository of) process models based on process scenarios. The method deals with retrieval of process models that describe (partially-specified) scenarios, where a process scenario is captured as a sequence of activity labels, cf. Section 3.3. In Section 4.1, we propose a formal

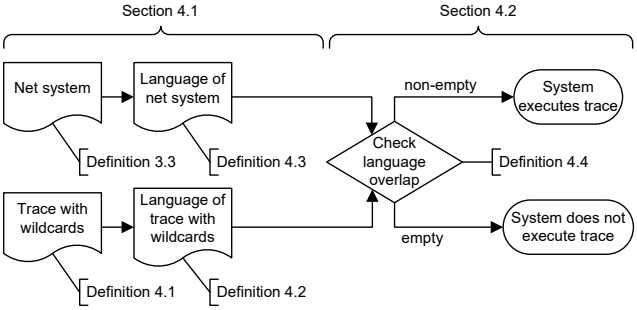


Figure 3: A flowchart that summarizes the trace executability problem.

model, called a *trace with wildcards*, which can be used to describe a collection of process scenarios. In addition, we explain how a system, cf. Definition 3.3, which is an example of a process model, can be interpreted as a collection of process scenarios. Then, Section 4.2 gives a rigorous definition of the *trace executability problem* that serves as a basis for the proposed in this paper method for scenario-based process querying. Figure 3 summarizes the main steps of the method.

4.1. Models for Process Scenarios

A trace with wildcards is a finite sequence in which each element is either a special wildcard character '$*$' or a pair composed of a label in $\mathbb{A}$ and a number between 0 and 1.

Definition 4.1 (Trace with wildcards)**.**
A *trace with wildcards* ω is a finite sequence over the set $\{*\} \cup (\mathbb{A} \times [0,1])$, where $* \notin \mathbb{A} \cup \{\tau, \gg\}$ is a special wildcard character. ⌟

For example, $\omega := \langle *, (\mathtt{a}, 1.0), (\mathtt{b}, 0.75), * \rangle$ is a trace with wildcards. By Ω, we denote the universe of all traces with wildcards. Every element of a trace with wildcards represents an *event*, or more precisely an abstract representation of an event. The precise interpretation of events in a trace with wildcards is specified using the meaning function with the signature $M_{\mathtt{Event}} : (\{*\} \cup \mathbb{A} \times [0,1]) \to \wp(\mathbb{A}^*)$, i.e., an event specifies a language. Let $sim : \mathbb{A} \times \mathbb{A} \to [0,1]$ be a label similarity function, i.e., a function that relates each pair of labels to a similarity score. Then, it holds that:

$$M_{\mathtt{Event}}(e) \;:=\; \begin{cases} \mathbb{A}^*, & e = * \\ \{a \in \mathbb{A} \mid sim(a, \pi_1(e)) \geq \pi_2(e)\}, & e \in \mathbb{A} \times [0,1]. \end{cases}$$

This definition assumes the existence of a *sim* function that assigns similarity scores to pairs of labels. A similarity score of zero signifies a pair of two incomparable labels, while a score of one identifies a pair of two identical labels. Two given labels are considered to be more similar if their similarity score, as per the *sim* function, is closer to one. One can rely on information retrieval methods to implement the *sim* function, cf. [60]. Hence, an event in a trace with wildcards represents either the set of all finite sequences composed of the labels in $\mathbb{A}$, if $e = *$, or the set of all activity labels that have a similarity score with label $\pi_1(e)$ that is at least $\pi_2(e)$, otherwise. In the former case, the event encodes the option to perform any sequence of activities. In the latter case, the

event stands for execution of one activity signified by the corresponding label. With all of the above, the meaning of a trace with wildcards is given as follows.

Definition 4.2 (Language of a trace with wildcards)**.**
The language of a trace with wildcards ω of length $n \in \mathbb{N}_0$ is denoted by $L(\omega)$ and is the concatenation of languages of its events, i.e., $L(\omega) := M_{\texttt{Event}}(\omega_{[1]}) \circ \ldots \circ M_{\texttt{Event}}(\omega_{[n]})$. ⌟

Let $\omega := \langle *, (\texttt{a}, 1.0), (\texttt{b}, 0.75), * \rangle$ be a trace with wildcards mentioned above such that $M_{\texttt{Event}}(\omega_{[2]}) = \{\texttt{a}\}$ and $M_{\texttt{Event}}(\omega_{[3]}) = \{\texttt{b}, \texttt{b}'\}$. Then, $L(\omega)$ is the language of all the strings that contain substrings `ab` or `ab`$'$. For example, $L(\omega)$ contains, among others, these strings: `ab`, `ab`$'$, `abcdhi`, `cccabccc`, and `abab`$\texttt{ab}'$`ab`. That is, ω represents the set of all process scenarios in which execution of activity `a` is directly followed either by activity `b` or by activity `b`$'$. Note that traces with wildcards specify languages that are in correspondence with the languages defined by the subclass of regular expressions that have one of the following forms:

1. ε,
2. $\{\texttt{a}\}$, $\texttt{a} \in \mathbb{A}$,
3. $(\mathbb{A}^*)$,
4. $(R \cup \{\texttt{a}\})$, $\texttt{a} \in \mathbb{A}$, and regular expression R derived using rules 2 and 4, or
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions derived using rules 1 through 5.

For example, if $M_{\texttt{Event}}((\texttt{a}, 1.0)) = \{\texttt{a}\}$, $M_{\texttt{Event}}((\texttt{b}, 1.0)) = \{\texttt{b}\}$, $M_{\texttt{Event}}((\texttt{a}, 0.5)) = \{\texttt{a}, \texttt{a}', \texttt{a}''\}$, and $M_{\texttt{Event}}((\texttt{b}, 0.75)) = \{\texttt{b}, \texttt{b}'\}$, then the correspondence is as follows:

$$
\begin{aligned}
L(\langle\rangle) &= \varepsilon, \\
L(\langle * \rangle) &= \mathbb{A}^*, \\
L(\langle (\texttt{a}, 1.0), (\texttt{b}, 1.0) \rangle) &= (\{\texttt{a}\} \circ \{\texttt{b}\}), \\
L(\langle *, (\texttt{a}, 1.0), (\texttt{b}, 0.75), * \rangle) &= ((((\mathbb{A}^*) \circ \{\texttt{a}\}) \circ (\{\texttt{b}\} \cup \{\texttt{b}'\})) \circ (\mathbb{A}^*)), \\
L(\langle (\texttt{a}, 0.5), *, * \rangle) &= (((((\{\texttt{a}\} \cup \{\texttt{a}'\}) \cup \{\texttt{a}''\}) \circ (\mathbb{A}^*)) \circ (\mathbb{A}^*)).
\end{aligned}
$$

Therefore, the language of a trace with wildcards is infinite if it contains at least one special wildcard character.

Finally, a net system specifies a language as follows.

Definition 4.3 (Language of a net system)**.**
The language of a net system $S := (N, M_{ini}, M_{fin})$, where $N := (P, T, F, \Lambda, \lambda)$, is denoted by $L(S)$ and is the set of strings $\{s \in \Lambda^* \mid \exists \sigma \in \mathbb{E}_S : \lambda(\sigma)|_\Lambda = s\}$. ⌟

Therefore, the language of a net system consists of the strings that can be written by recording labels of observable transitions in all the executions of the system, also called *label sequences* of the system. Note that the language of a net system is infinite in case the system specifies cyclic dependencies on its transition occurrences. Every string in the language encodes a sequence of activity labels observed in the corresponding process scenario. For example, the language of the system in Figure 1 is infinite and contains, among others, these strings: `abcdhi`, `acbdhi`, `abcdfgi`, `acbdfgi`, and `abcdecbdecbdecbdecbdebcdecbdegfi`.

4.2. The Trace Executability Problem

In this section, we propose the trace executability problem which given a process model and a scenario (template) consists of deciding whether the model describes the scenario (a scenario that matches the template). The problem is instantiated for net systems and traces (with wildcards) as example formalisms for specifying process models and scenario templates, respectively. However, it is straight-forward to adapt the problem to other similar formalisms.

A net system *executes* a trace if there exists an execution of the system that induces the trace. Formally, this is specified as a check of whether there exists a string that belongs to both languages, the language of the net system and the language of the trace.

Definition 4.4 (Trace executability)**.**
A net system $S \in \mathbb{S}$ *executes a trace with wildcards* ω, denoted by *executes*(S, ω), iff $L(S) \cap L(\omega) \neq \emptyset$. ⌟

For example, the net system in Figure 1 executes $\omega := \langle *, (\mathtt{a}, 1.0), (\mathtt{b}, 0.75), * \rangle$ because their languages both contain the string `abcdhi`.

Note that the above definition does not require that $L(\omega) \subseteq L(S)$. If that was the case, then every check of whether a net system with a finite language executes an infinite language (a trace with a wildcard character) would evaluate to false. The proposed definition does not suffer from this problem. At the same time, if one wants to check whether a net system executes a finite language, i.e., every process scenario in a given finite set of process scenarios, then one can still rely on Definition 4.4 to check if the net system indeed executes every single scenario in the given set.

Finally, given a trace with wildcards and a repository of process models, the proposed scenario-based process querying method retrieves all the process models from the repository that execute the trace.

5. Process Query Language

PQL is a domain-specific programming language for managing process models based on scenarios that these models describe. This section specifies the part of PQL that implements scenario-based process querying method proposed in Section 4. It defines the syntactic and semantic rules of PQL using the notation introduced in [61]. Sections 5.1, 5.2, and 5.3, respectively, present the abstract syntax, concrete syntax, and the dynamic semantics of the scenario-based extension of PQL, whereas Section 5.4 shows and explains some sample PQL queries. Note that the fragment of PQL proposed in this section is self-contained, i.e., it can be used as a stand-alone language for implementing scenario-based process querying.[5] A rigorous description of the remaining part of PQL can be found in [36].

[5]The most recent complete grammar of PQL which includes the extension proposed in this paper can be accessed via `https://github.com/processquerying/PQL/blob/master/antlr/PQL.g4`.

5.1. Abstract Syntax

The abstract syntax of PQL defines its core structure. It neither commits to specific choices for keywords, nor fixes the order of various statements of the language. According to the notation in [61], an abstract syntax of a language can be given as a finite set of *constructs* and a finite set of *productions* associated with the constructs. A construct describes the structure of a *specimen* of the language using productions of three types: *aggregate*, *choice*, and *list* productions. Note that the precise meaning of all the proposed constructs is proposed in Section 5.3.

The core structure of all PQL programs is captured by the `Query` construct.

$$\texttt{Query} \triangleq \textit{atts} : \texttt{Attributes}; \textit{locs} : \texttt{Locations}; \textit{pred} : \texttt{Predicate}$$

The `Query` construct is defined as an *aggregate* production composed of three components. In general, an aggregate production defines a construct made of a fixed number of components that are separated by semicolons. Each component is preceded by a *tag* that indicates its *role* within the construct. Therefore, every PQL query is composed of attributes, locations, and a predicate, which are distinguished via tags *atts*, *locs*, and *pred*, respectively.

Both the `Attributes` construct and the `Locations` construct are defined as choice productions that offer two alternatives. In general, a choice production specifies the corresponding construct as a collection of alternatives which are separated by vertical bar symbols. Both `Attributes` and `Locations` can be defined as the `Universe` construct, to refer to all known attributes and locations, respectively. Alternatively, `Attributes` and `Locations` can be given as `ListOfAttributes` and `ListOfLocations`, respectively.

$$\begin{aligned} \texttt{Attributes} &\triangleq \texttt{Universe} \mid \texttt{ListOfAttributes} \\ \texttt{Locations} &\triangleq \texttt{Universe} \mid \texttt{ListOfLocations} \end{aligned}$$

$$\begin{aligned} \texttt{ListOfAttributes} &\triangleq \texttt{Attribute}^{+} \\ \texttt{ListOfLocations} &\triangleq \texttt{Location}^{+} \end{aligned}$$

Both `ListOfAttributes` and `ListOfLocations` are *list* productions. In general, a list production defines a sequence of zero, one, or more specimens of another construct. Every list of attributes must contain at least one attribute specimen, denoted by $\texttt{Attribute}^{+}$. Similarly, every sequence of locations must contain at least one location specimen.

A PQL predicate is defined as a choice production with four alternatives.

$$\begin{aligned} \texttt{Predicate} &\triangleq \texttt{Negation} \mid \texttt{Conjunction} \mid \texttt{Disjunction} \mid \texttt{Executes} \\ \texttt{Negation} &\triangleq \textit{pred} : \texttt{Predicate} \\ \texttt{Conjunction} &\triangleq \textit{pred}_1 : \texttt{Predicate}; \textit{pred}_2 : \texttt{Predicate} \\ \texttt{Disjunction} &\triangleq \textit{pred}_1 : \texttt{Predicate}; \textit{pred}_2 : \texttt{Predicate} \\ \texttt{Executes} &\triangleq \textit{tr} : \texttt{Trace} \end{aligned}$$

Every `Negation` and every `Executes` predicate are defined using a single specimen of the `Predicate` and `Trace` construct, respectively. A `Conjunction` and `Disjunction` predicate are aggregations of two specimens of `Predicate`.

The `Trace` construct is defined below.

$$
\begin{array}{rcl}
\texttt{Trace} & \triangleq & \texttt{Event}^{*} \\
\texttt{Event} & \triangleq & \texttt{Universe} \mid \texttt{Task} \\
\texttt{Task} & \triangleq & \texttt{ExactTask} \mid \texttt{DefSimTask} \mid \texttt{SimTask} \\
\texttt{ExactTask} & \triangleq & \textit{label} : \texttt{Label} \\
\texttt{DefSimTask} & \triangleq & \textit{label} : \texttt{Label} \\
\texttt{SimTask} & \triangleq & \textit{label} : \texttt{Label};\ \textit{sim} : \texttt{Similarity}
\end{array}
$$

Hence, a specimen of `Trace` is a sequence of zero or more specimens of `Event`; the asterisk symbol stands for the *Kleene star*—with its standard language theory meaning. A specimen of `Event` is either a specimen of `Universe` or `Task`. PQL offers three ways to specify a task, which are detailed above.

5.2. Concrete Syntax

This section presents an extract of the machine- and human-readable encoding of PQL for the scenario-based querying. As mentioned above, the concrete syntax of PQL is inspired by that of SQL—a programming language for managing data stored in a relational database management system (DBMS) [62]. This decision is due to the fact that the syntax of SQL is well-recognized and that, despite addressing a different domain, i.e., dynamic processes versus static data, PQL aims to serve a similar purpose, querying for information. Note that one can propose a different concrete syntax of PQL as a mapping from its abstract syntax to some specific encoding.

We define the SQL-like concrete syntax of PQL by giving, for each of the constructs of the abstract grammar (except those defined using choice productions), a function which takes that construct as input and yields all its specific forms. We denote each such function by the name of the respective construct of the abstract grammar with appended subscript `c`. We start by suggesting a function for the topmost construct and proceed by gradually refining its components. Below, the reader can find the function that defines all the possible concrete encodings of the `Query` construct.

$$
\begin{array}{rcl}
\texttt{Query}_{\texttt{c}}(q : \texttt{Query}) & \triangleq & \texttt{`SELECT'}\ \texttt{Attributes}_{\texttt{c}}(q.\textit{atts}) \\
 & & \texttt{`FROM'}\ \texttt{Locations}_{\texttt{c}}(q.\textit{locs}) \\
 & & (\texttt{`WHERE'}\ \texttt{Predicate}_{\texttt{c}}(q.\textit{pred}))?\ \texttt{`;'}
\end{array}
$$

Thus, a specimen of the `Query` construct can be encoded as a character string that starts with the `SELECT` keyword, followed by the concrete encoding of the attributes, followed by the `FROM` keyword, followed by the concrete encoding of the locations, which can be followed by the `WHERE` keyword and the concrete encoding of the predicate, followed by the semicolon `';'`. There can be an arbitrary number of whitespace characters between any two subsequent components of the string. The order of various components is fixed. The astute reader may have already discerned that we use regular expressions to define the concrete syntax of PQL.

The concrete syntax of the `Universe` construct is the asterisk symbol `'*'`. A concrete encoding of the `ListOfAttributes` construct is a comma separated sequence of

encodings of its constituting attributes, where a concrete encoding of the `Attribute` construct is a character string enclosed in double quotes.

$$\begin{aligned}\texttt{ListOfAttributes}_\texttt{C}(loa:\ \texttt{ListOfAttributes}) \triangleq\ & isEmpty(loa)\ ?\ \text{`'}\ :\ \texttt{Attribute}_\texttt{C}(loa.FIRST)\\ & (isEmpty(loa.TAIL)\ ?\ \text{`'}\ :\ \text{`,'})\\ & \texttt{ListOfAttributes}_\texttt{C}(loa.TAIL)\end{aligned}$$

The regular expression that defines possible encodings of the `ListOfLocations` construct is similar to the regular expression given above, i.e., it defines all comma separated lists of locations, where a concrete encoding of the `Location` construct is a character string enclosed in double quotes.

Next, we propose the concrete syntax of all the alternatives associated with the `Predicate` construct.

$$\texttt{Negation}_\texttt{C}(p:\texttt{Negation}) \triangleq \text{`NOT'}\ \texttt{Predicate}_\texttt{C}(p.pred)$$

$$\begin{aligned}\texttt{Conjunction}_\texttt{C}(p:\texttt{Conjunction}) &\triangleq \texttt{Predicate}_\texttt{C}(p.pred_1)\ \text{`AND'}\ \texttt{Predicate}_\texttt{C}(p.pred_2)\\ \texttt{Disjunction}_\texttt{C}(p:\texttt{Disjunction}) &\triangleq \texttt{Predicate}_\texttt{C}(p.pred_1)\ \text{`OR'}\ \texttt{Predicate}_\texttt{C}(p.pred_2)\end{aligned}$$

A concrete encoding of the `Executes` predicate is a character string that starts with `'Executes'` and is followed by an encoding of the trace enclosed in parentheses.

$$\texttt{Executes}_\texttt{C}(p:\texttt{Executes}) \triangleq \text{`Executes'}\ \text{`('}\ \texttt{Trace}_\texttt{C}(p.tr)\ \text{`)'}$$

Though not specified explicitly, every PQL predicate can be enclosed in parentheses.

A concrete encoding of the `Trace` construct is as a comma separated sequence of encodings of its constituting events put within angle brackets, as specified below.

$$\begin{aligned}\texttt{Trace}_\texttt{C}(t:\texttt{Trace}) &\triangleq \text{`<'}\ \texttt{Events}_\texttt{C}(t)\ \text{`>'}\\ \texttt{Events}_\texttt{C}(t:\texttt{Trace}) &\triangleq isEmpty(t)\ ?\ \text{`'}\ :\ \texttt{Event}_\texttt{C}(t.FIRST)\\ &\quad (isEmpty(t.TAIL)\ ?\ \text{`'}\ :\ \text{`,'})\ \texttt{Trace}_\texttt{C}(t.TAIL)\end{aligned}$$

An event in a trace can be specified either using the `Universe` or `Task` construct. As the `Universe` construct is defined above, next we propose three encodings of a task.

$$\begin{aligned}\texttt{ExactTask}_\texttt{C}(t:\texttt{ExactTask}) &\triangleq \text{`"'}\ \texttt{Label}_\texttt{C}(t.label)\ \text{`"'}\\ \texttt{DefSimTask}_\texttt{C}(t:\texttt{DefSimTask}) &\triangleq \text{`\~{}'}\ \text{`"'}\ \texttt{Label}_\texttt{C}(t.label)\ \text{`"'}\\ \texttt{SimTask}_\texttt{C}(t:\texttt{SimTask}) &\triangleq \text{`"'}\ \texttt{Label}_\texttt{C}(t.label)\ \text{`"'}\ \text{`['}\ \texttt{Similarity}_\texttt{C}(t.sim)\ \text{`]'}\end{aligned}$$

A concrete encoding of the `Label` construct is any character string. Thus, a PQL task is a string put within the quotation marks (an exact task `ExactTask`), which also may be preceded by the tilde grapheme (a default similarity task `DefSimTask`) or succeeded by a number between (and including) zero and one written in the square brackets (a similarity task `SimTask`); a concrete encoding of the `Similarity` construct is any character string that can be interpreted as a decimal representation of a real number greater or equal to zero and less than or equal to one, e.g., 0.5, .95, and 1.0.

The string '`SELECT * FROM * WHERE Executes(<a,*,d,e,*,~j,f,*>);`' is an example of a concrete encoding of a PQL query. In addition to the short labels in the caption of Figure 1, this query uses short label `j` := "archive booking request". The query specifies attributes and locations using the `Universe` construct, and the predicate is given using the `Executes` predicate with the `Trace` construct defined as a sequence of eight events, three `Universe` events, four `ExectTask` events, and one `DefSimTask` event.

5.3. Dynamic Semantics

The meaning functions of PQL specify the effects of its valid constructs using mathematical denotations. These denotations are defined over the following domains:

- $\mathbb{A}$, a universe of *attribute names*;
- $\mathbb{B}$, a universe of *attribute values*;
- $\mathbb{L}$, a universe of *locations*;
- $\mathbb{S}$, a universe of *net systems*;
- $\mathbb{T} := \wp_{\geq 1}(\mathbb{C})$, the universe of all *tasks* over the universe of all character strings $\mathbb{C}$.[6]

Given $\mathbb{A}$ and $\mathbb{B}$, we assume that there exists a function $\chi : \mathbb{A} \to \wp_{\geq 1}(\mathbb{B})$ that maps attribute names onto sets of permissible attribute values.

Every PQL query is evaluated in the context of a repository of net systems. To permit subsequent formal discussions, we give a rigorous definition of a repository.

Definition 5.1 (Repository)**.**
A *process repository*, or a *repository*, is a 6-tuple $R := (S, A, L, val, loc, \precsim)$, where $S \subseteq \mathbb{S}$ is a finite set of *net systems*, $A \subseteq \mathbb{A}$ is a finite set of *attribute names*, $L \subseteq \mathbb{L}$ is a set of *locations*, $val : S \times A \to \mathbb{B}$ is the *attribute value assignment* function, such that $\forall (s,a) \in S \times A : val(s,a) \in \chi(a)$, $loc : S \to L$ is the *location assignment* function, and $\precsim$ is a reflexive binary relation over L, called *location map*. ⌟

A repository is thus a collection of net systems that are associated with their attribute values, e.g., authors, versions, and times of creation of net systems (via the attribute value assignment function), and locations, e.g., folders (via the location assignment function), where the net systems are stored. The location map of a repository defines a containment relation over locations, where a location $l_1 \in L$ *is contained in* a location $l_2 \in L$ iff it holds that $l_1 \precsim l_2$. By `Repository`, we denote the universe of repositories.

In what follows, for a selection of PQL constructs that relate to the scenario-based querying, we propose meaning functions that explain the meaning of specimens of those constructs. We adopt the notation in which function $M_{\texttt{Construct}}$ defines the meaning of construct `Construct`; note that $M_{\texttt{Rel}}$ is an auxiliary meaning function and does not relate to a particular construct.

$$M_{\texttt{Rel}} \quad : \quad \texttt{Query} \times \texttt{Repository} \to \wp(\mathbb{S})$$

$$M_{\texttt{Rel}}[q : \texttt{Query}, (S, A, L, val, loc, \precsim) : \texttt{Repository}] \quad \triangleq \quad \{s \in S \mid (\exists\, l \in M_{\texttt{Locations}}(q.locs) : loc(s) \precsim l) \wedge M_{\texttt{Predicat}}(q.pred, s)\}$$

[6] For a set A, $\wp_{\geq 1}(A)$ is the set of all non-empty subsets of A, i.e., the power set of A without the empty set.

Given a specimen of `Query` and a repository, $M_{\texttt{Rel}}$ returns the set of net systems that are *relevant* for the purpose of the query, i.e., systems that are contained in the locations of interest and satisfy the predicate.

The meaning function of the `Query` construct can now be specified as follows.

$$M_{\texttt{Query}} \quad : \quad \texttt{Repository} \times \texttt{Query} \to \texttt{Repository}$$

$M_{\texttt{Query}}[r := (S,A,L,val,loc,\precsim) : \texttt{Repository}, q : \texttt{Query}] \triangleq (S',A',L,val',loc',\precsim) \in \texttt{Repository}$, where $S' := M_{\texttt{Rel}}(q,r)$, $A' := M_{\texttt{Attributes}}(q.atts) \cap A$, $val' := val|_{(S' \times A')}$, and $loc' := loc|_{S'}$.

Thus, the meaning of a PQL query q in the context of a repository r is a special 'projection' of r, i.e., a repository of systems that are relevant for the purpose of the query together with values of those of their attributes specified by $q.atts$.

The meaning of a specimen of the `Attributes` construct is a set of attributes.

$$M_{\texttt{Attributes}} \quad : \quad \texttt{Attributes} \to \wp(\mathbb{A})$$

$$M_{\texttt{Attributes}}[as : \texttt{Attributes}] \quad \triangleq \quad \begin{cases} \mathbb{A} & as \text{ is } \texttt{Universe} \\ M_{\texttt{ListOfAttributes}}(as) & otherwise \end{cases}$$

$$M_{\texttt{ListOfAttributes}} \quad : \quad \texttt{ListOfAttributes} \to \wp(\mathbb{A})$$

$$M_{\texttt{ListOfAttributes}}[loa : \texttt{ListOfAttributes}] \quad \triangleq \quad \bigcup_{i \in [1..|loa|]} M_{\texttt{Attribute}}(loa_{[i]})$$

The meaning of a specimen of `Locations` is defined similarly to the meaning of a specimen of `Attributes`. To obtain the definition of the meaning function for the `Locations` construct, in the above functions, one needs to replace `Attributes`, `ListOfAttributes`, and $\mathbb{A}$, with `Locations`, `ListOfLocations`, and $\mathbb{L}$, respectively.

Next, we detail the meanings of several predicate types.

$$\begin{aligned} M_{\texttt{Negation}}[p : \texttt{Negation}, s : \mathbb{S}] &\triangleq \neg\, M_{\texttt{Predicate}}(p.pred, s) \\ M_{\texttt{Conjunction}}[p : \texttt{Conjunction}, s : \mathbb{S}] &\triangleq M_{\texttt{Predicate}}(p.pred_1, s) \wedge \\ &\qquad M_{\texttt{Predicate}}(p.pred_2, s) \\ M_{\texttt{Disjunction}}[p : \texttt{Disjunction}, s : \mathbb{S}] &\triangleq M_{\texttt{Predicate}}(p.pred_1, s) \vee \\ &\qquad M_{\texttt{Predicate}}(p.pred_2, s) \\ M_{\texttt{Executes}}[p : \texttt{Executes}, s : \mathbb{S}] &\triangleq executes(s, M_{\texttt{Trace}}(p.tr)) \end{aligned}$$

PQL supports logical expressions using standard connectives of negation, conjunction, and disjunction, that can be captured as specimens of the `Negation`, `Conjunction`, and `Disjunction` construct, respectively. The fact that PQL supports negation and conjunction makes its logical expressions functionally complete. PQL suggests the following precedence of the logical operations: brackets "()", then negation "¬", then conjunction "∧", and finally disjunction "∨". Thus, the expression $a \vee \neg b \wedge (c \vee d) \wedge e$ must be evaluated as $(a \vee ((\neg b) \wedge (c \vee d) \wedge e))$.

Finally, a specimen of `Trace` specifies a trace with wildcards.[7]

$$M_{\texttt{Trace}} \quad : \quad \texttt{Trace} \to \Omega$$

$$M_{\texttt{Trace}}[tr : \texttt{Trace}] \quad \triangleq \quad isEmpty(tr) \ ? \ \langle\rangle \ : \ \langle M_{\texttt{Element}}(tr.FIRST)\rangle \circ M_{\texttt{Trace}}(tr.TAIL)$$

$$M_{\texttt{Element}} \quad : \quad \texttt{Event} \to \{*\} \cup (\mathbb{\Lambda} \times [0,1])$$

$$M_{\texttt{Element}}[e : \texttt{Event}] \triangleq \begin{cases} * & e \texttt{ is Universe} \\ (e.label, 1.0) & e \texttt{ is ExactTask} \\ (e.label, defaultSim) & e \texttt{ is DefSimTask} \\ (e.label, e.sim) & e \texttt{ is SimTask} \end{cases}$$

Thus, the PQL query '`SELECT * FROM * WHERE Executes(<a,*,d,e,*,~j,f,*>);`' specifies an intent to retrieve every net system S from a given repository where the language of S contains a string that starts with `a`, followed by `d` which is immediately followed by `e`, followed by `j` or a similar label which is immediately followed by `f`.

5.4. Sample Queries

This section exemplifies the proposed scenario-based process querying method using six process models and ten queries. Figure 4 depicts the models. They are the adapted versions of models from [63, 64] captured in BPMN [65]. For simplicity, the models use abstract activity labels. Activities are drawn as rectangles with rounded corners. Gateways are visualized as diamonds. Exclusive gateways use a marker which is shaped like "×" inside the diamond shape, wheres parallel gateways use a marker which is shaped like "+" inside the diamond shape. Directed arcs encode control flow dependencies. Models 1 and 4 are acyclic. The other four models (models 2, 3, 5, and 6) contain cyclic paths. Models 1, 2, and 6 are well-structured, while the other models (models 3, 4, and 5) are unstructured.[8] Note that though model 5 is unstructured, there exists a well-structured model that is equivalent to model 5. This is model 6 in Figure 4. Thus, models 5 and 6 induce the same language.

It is a common practice to define the execution semantics of business process models through their translations to Petri nets. Then, it is accepted that a model behaves according to the execution principles of the corresponding net system. Accordingly, to evaluate PQL queries over the BPMN models from Figure 4, we translate them to corresponding Petri nets. The fact that the dynamic semantics of PQL queries is defined over Petri nets makes it suitable for evaluation over models captured in process modeling languages for which the translations to Petri nets exist. For example, translations of models captured in BPMN, EPC, BPEL, and UML Activity Diagrams to Petri nets

[7] We assume that there exists a global constant $defaultSim \in [0,1]$ that specifies the default label similarity threshold.

[8] Intuitively, a process model is *well-structured* if and only if every gateway with multiple outgoing arcs (a split) has a corresponding gateway with multiple incoming arcs (a join), and vice versa, such that the set of nodes between the split and the join induces a single-entry-single-exit (SESE) region; otherwise the process model is *unstructured* [63].

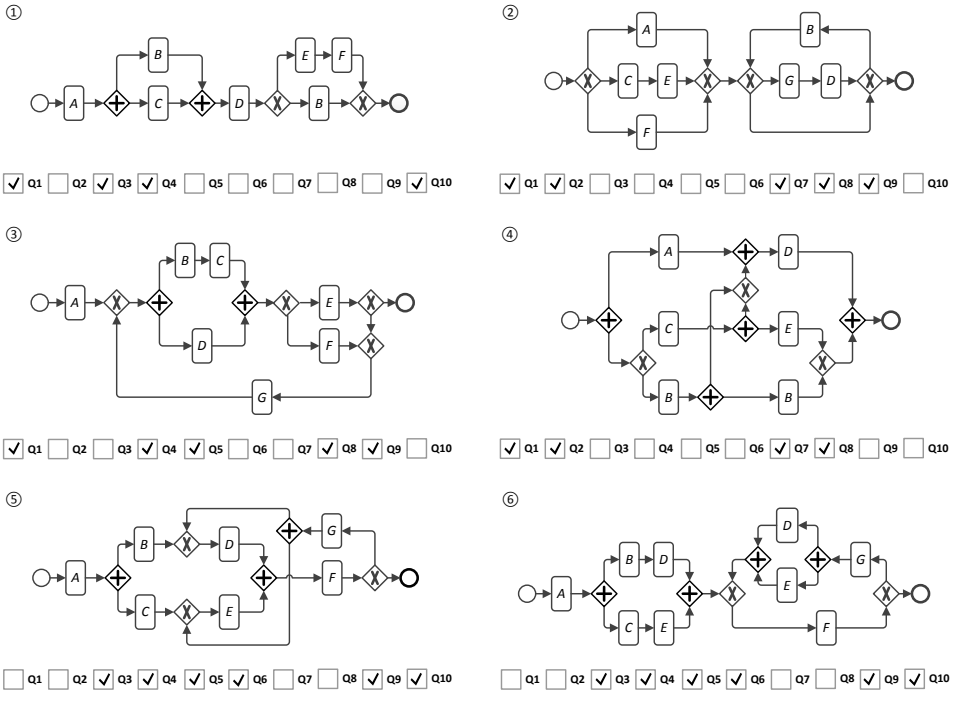


Figure 4: Six sample process models captured using BPMN and results of interpreting ten sample PQL queries from Section 5.4 on these models.

are readily available. We used the approach from [66] to accomplish the translation of BPMN models to Petri nets.

Next, we define ten sample queries as instantiations of the template $Q :=$ `'SELECT * FROM * WHERE Executes(`ω`);'`, where ω is a placeholder for concrete encodings of traces with wildcards. Ten sample encodings of traces are listed below:

$\omega_1 :=$ `'<*,B,*,D,*,B,*>';`
$\omega_2 :=$ `'<A,*,D>';`
$\omega_3 :=$ `'<*,D,E,F>';`
$\omega_4 :=$ `'<A,B,C,*>';`
$\omega_5 :=$ `'<*,D,E,*,D,E,*>';`
$\omega_6 :=$ `'<*,E,D,*,E,D,*>';`
$\omega_7 :=$ `'<*,B,B,*>';`
$\omega_8 :=$ `'<*,B,*,*,B,*,D,*>';`
$\omega_9 :=$ `'<*,G,*>';`
$\omega_{10} :=$ `'<A,B,C,D,E,F>';`

Thus, sample query Q_i, $i \in [1..10]$, is obtained by replacing ω in the template with ω_i. For example, query $Q_1 :=$ `'SELECT * FROM * WHERE Executes(<*,B,*,D,*,B,*>);'`, $Q_2 :=$ `'SELECT * FROM * WHERE Executes(<A,*,D>);'`, etc.

Figure 4 also depicts semantics, i.e., the meaning, of the ten sample queries in the context of the sample repository. The meaning of a query is the repository composed of every model for which the query name under that model is ticked. For example, the meaning of Q_1 is the repository composed of models 1, 2, 3, and 4, whereas the meaning of query Q_5 is the repository composed of models 3, 5, and 6, etc. Note that since models 5 and 6 induce the same language, any query evaluation over these

models yields the same result. Note also that the meaning of each sample query over the BPMN models was derived over corresponding net systems obtained using the translation approach proposed in [66].

To illustrate why certain models are included in the result of a query, consider query Q_1. Model 1 is part of the repository that results from interpreting Q_1 because string ABCDB is contained in both languages, the language of model 1 and $L(M_{\texttt{Trace}}(\upsilon))$, where $\omega_1 \in \texttt{Trace}_{\texttt{C}}(\upsilon)$, $\upsilon \in \Omega$. Similarly, strings that can be used to justify inclusion of models 2, 3, and 4 in the repository that results from evaluation of Q_1 are CEBGDBB, ABDCFGDBCE, and ACBDBE, respectively. This example demonstrates that when evaluating a query, a decision on whether to include a model into a resulting repository cannot be carried out based on a check of a single structural property of the model; or, more precisely, it is not known if such a property, expressed in as a finite statement, exists. In model 1, there exists a directed path that visits activity *B*, then activity *D*, and then again activity *B*. However, in model 4, there is no such path.

Consider this query that contains three logical operators: '`SELECT * FROM * WHERE (Executes(`ω_7`) OR Executes(`ω_{10}`)) AND NOT Executes(`ω_6`);`'. The result of this query is the repository that comprises models 1, 2, and 4: predicate `Executes(`ω_6`)` evaluates to FALSE for all of them (hence, the negation of the predicate evaluates to TRUE), and `Executes(`ω_{10}`)` evaluates to TRUE for model 1, while `Executes(`ω_7`)` evaluates to TRUE for models 2 and 4. Model 3 is not included in the result as both `Executes(`ω_7`)` and `Executes(`ω_{10}`)` evaluate to FALSE for this model, while models 5 and 6 are excluded because predicate `Executes(`ω_6`)` evaluates to TRUE for them.

6. Deciding Trace Executability

The previous section rigorously defines all the components of the proposed PQL extension for supporting scenario-based process querying except for an approach to decide the trace executability problem captured in Definition 4.4. This section closes this gap by proposing a technique for computing whether a given workflow system executes a trace with wildcards. Workflow systems were introduced as a subclass of net systems that are particularly suitable for describing business processes [55]. Hence, the approach presented in this section aims to support the querying of process models of business processes commonly developed and maintained in organizations. As industrial workflow nets often have syntactic and semantic errors [4], we also present some initial results for deciding the trace executability problem over net systems.

Figure 5 shows a flowchart that summarizes our scenario-based process querying technique. Given a workflow system *S* and a trace with wildcards ω, the decision procedure starts by performing three transformations of *S* informed by ω, namely sets of labels unification, framing, and sequences test insertion; refer to Section 6.1 for details. Note that our solution to deciding the trace executability problem relies on the checks over the augmented system obtained after these transformations. Unification of labels addresses the problem of multiple transitions in the system with the same label. After this transformation, for each label in the query trace, there exists exactly one transition in the system that should be matched with it. Framing of the system ensures the existence of dedicated transitions that appear at the start and end of each its execution. These transitions are necessary to correctly process queries that do not

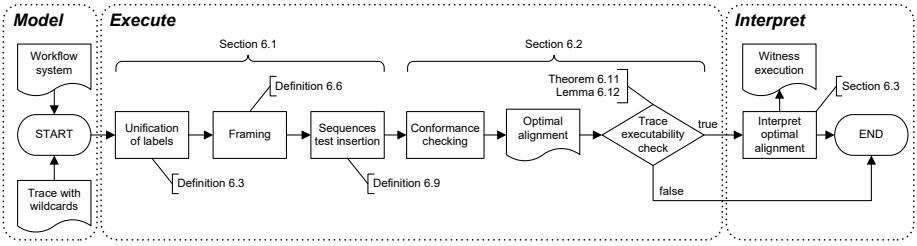


Figure 5: A flowchart that summarizes scenario-based process querying technique.

start/end with the wildcard character. Finally, a sequence test transformation inserts transitions that represent occurrences of maximal label subsequences from the corresponding query trace. They are used to ensure that the system can execute these label subsequences without performing any other activities. Then, an optimal alignment is computed between the transformed system and a special trace derived from ω. The trace executability check is performed based on the cost of the constructed optimal alignment; refer to Section 6.2 for details. Finally, Section 6.3 proposes an approach for explaining why the input system S executes ω by constructing a so-called *witness execution* of the system.

Note that the steps of the flowchart in Figure 5 are grouped according to the *model*, *execute*, and *interpret* parts of the process querying framework [33].

6.1. Useful Transformations of Net Systems

This section presents three transformations of net systems, namely *label unification*, *framing*, and *sequence test insertion*, and discusses their useful properties.

Unification of labels. The *label unification* transformation of a net system for a given label proposed in [67] can be used to transform the system into a fresh system that exhibits equivalent behavior to the original system but in which all observations of the given label are guaranteed to be triggered by occurrences of a single dedicated transition. Next, we generalize this principle to accept a set of labels as input.

Let $N := (P, T, F, \Lambda, \lambda)$ be a net and let $\Delta \subseteq \mathbb{A}$ be a set of labels. By $trs(N, \Delta)$ we denote the set of all transitions of N that have a label that is contained in Δ, i.e., $trs(N, \Delta) := \{t \in T \mid \lambda(t) \in \Delta\}$. Then, *labels unification* is defined as follows.

Definition 6.1 (Labels unification)**.**
A result of *labels unification* in a net system $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, for a set of labels $\Delta \subseteq \mathbb{A}$, $X := trs(N, \Delta)$, is a pair $(\hat{S}, \alpha)$ such that:
1. $\hat{S} = S$ and $\alpha \in \mathbb{A} \setminus \Lambda$, if $X = \emptyset$,
2. $\hat{S} = S$ and $\alpha = \lambda(t)$, if $X = \{t\}$, $t \in T$, or
3. $\hat{S} = (\hat{N}, M_{ini}, M_{fin})$, $\hat{N} := (\hat{P}, \hat{T}, \hat{F}, \hat{\Lambda}, \hat{\lambda})$, and $\alpha \in \mathbb{A} \setminus \Lambda$:
   - $\hat{P} := P \cup \{\acute{p}^\alpha, \grave{p}^\alpha\} \cup (\bigcup_{t \in X} \{p_t^\alpha\})$,
   - $\hat{T} := T \cup \{\hat{t}^\alpha\} \cup (\bigcup_{t \in X} \{\acute{t}^\alpha, \grave{t}^\alpha\})$,
   - $\hat{F} := F \cup \{(\acute{p}^\alpha, \hat{t}^\alpha), (\hat{t}^\alpha, \grave{p}^\alpha)\} \cup (\bigcup_{t \in X} \{(p, \acute{t}^\alpha) \mid p \in {\bullet t}\}) \cup$
     $(\bigcup_{t \in X} \{(\acute{t}^\alpha, p_t^\alpha), (p_t^\alpha, \grave{t}^\alpha), (\acute{t}^\alpha, \acute{p}^\alpha), (\grave{p}^\alpha, \grave{t}^\alpha)\}) \cup$
     $(\bigcup_{t \in X} \{(\grave{t}^\alpha, p) \mid p \in t{\bullet}\})$,
   - $\hat{\Lambda} := \Lambda \cup \{\alpha\}$, and

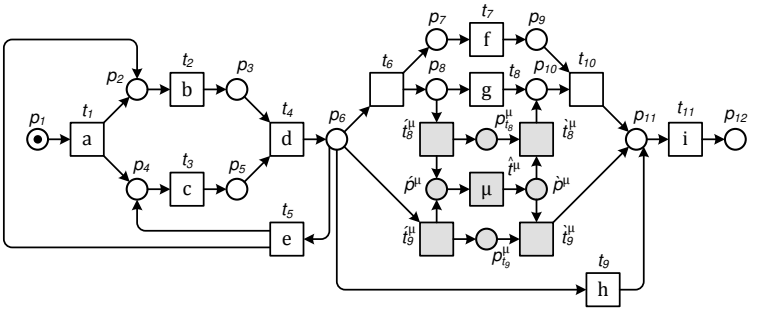


Figure 6: A result of *labels unification* in the net system in Figure 1 for $\{\mathtt{g},\mathtt{h}\}$.

– $\hat{\lambda} := \lambda \cup \{(\hat{t}^{\alpha}, \alpha)\} \cup \{(\acute{t}^{\alpha}, \tau) \mid t \in X\} \cup \{(\grave{t}^{\alpha}, \tau) \mid t \in X\}$.

Note that $(\{\acute{p}^{\alpha}, \grave{p}^{\alpha}, \hat{t}^{\alpha}\} \cup (\bigcup_{t \in X} \{p_t^{\alpha}, \acute{t}^{\alpha}, \grave{t}^{\alpha}\})) \cap (P \cup T) = \varnothing$, and $\hat{P}$, $\hat{T}$, $\mathbb{\Lambda}$ are pairwise disjoint. ⌟

The fresh transition $\hat{t}^{\alpha}$ has a unique label in $\hat{S}$ and is called the *solitary* transition for labels Δ. Accordingly, fresh transitions $\acute{t}^{\alpha}$ and $\grave{t}^{\alpha}$ are called *presolitary* and *postsolitary* transitions of t for α. Figure 6 shows a result of *labels unification* (S', μ) in the net system S of Figure 1 for the set of labels $\{\mathtt{g},\mathtt{h}\}$; the fresh places and transitions are highlighted with gray background. The label unification transformation proposed in [67] forbids occurrences of transitions that have the input label in the resulting system. According to its generalized version proposed above, transitions that have labels from the input set can participate in occurrence sequences of the resulting system. The transformed system in Figure 6, similar to the system in Figure 1, is a workflow system.

Proposition 6.2 (Unification in workflow system)**.**
If (S, α) is a result of labels unification in a workflow system for a set of labels, then S is a workflow system. ⌟

The proof of Proposition 6.2 follows immediately from the definition of a workflow system and Definition 6.1.

One can perform several labels unifications in a given net system as follows.

Definition 6.3 (Sets of labels unification)**.**
A result of *sets of labels unification* in a net system $S_0 := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, for $\{\Delta_1, \ldots, \Delta_n\} \subseteq \wp(\mathbb{\Lambda})$, $n \in \mathbb{N}_0$, is a pair (S_n, g) such that there exists a sequence $\langle (S_1, \alpha_1), \ldots, (S_n, \alpha_n) \rangle$ where each element at position $i \in [1..n]$ is a result of labels unification in S_{i-1} for Δ_i, and $g := \bigcup_{i \in [1..n]} \{(\alpha_i, \Delta_i)\}$. ⌟

A result of sets of labels unification in a net system S_0 for a set of sets of labels Δ is a pair composed of the transformed system S_n and a bijective function that maps every unique label of a fresh solitary transition in S_n onto the set of labels in Δ that was used to introduce the solitary transition. E.g., $(S', \{(\mu, \{\mathtt{g},\mathtt{h}\})\})$, where S' is the system in Figure 6, is a result of sets of labels unification in the system from Figure 1 for $\{\{\mathtt{g},\mathtt{h}\}\}$.

If S_0 is a workflow system, by induction on the sequence of labels unifications, it follows that S_n is a workflow system.

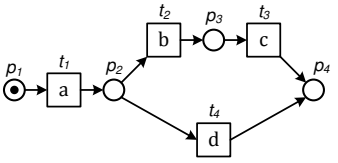


Figure 7: A net system.

Corollary 6.4 (Unification in workflow system)**.**
If (S,g) is a result of sets of labels unification in a workflow system for $\Delta \subseteq \wp(\mathbb{A})$, then S is a workflow system. ⌟

A net system that stems from a (sets of) labels unification in a net system S is in a strong behavioral equivalence with S, i.e., their languages are closely related. Let A be an alphabet. Then, $expand : A^* \times (A \nrightarrow \wp(A)) \rightarrow \wp(A^*)$ is given by $expand(\rho, g) := \{\eta \in A^* \mid (|\rho| = |\eta|) \wedge (\forall i \in [1\,..\,|\eta|] : ((\eta_{[i]} = \rho_{[i]}) \vee (\eta_{[i]} \in dom(g) \wedge \rho_{[i]} \in g(\eta_{[i]}))))\}$, where $\rho \in A^*$ and $g : A \nrightarrow \wp(A)$. That is, given a string ρ over an alphabet A and a function g that maps some symbols from A onto sets of symbols from A, $expand(\rho, g)$ is the set that contains every string of the same length as ρ in which the symbol at position i is either the same as the symbol of ρ at position i or is a symbol x from the domain of g such that $\rho_{[i]} \in g(x)$.

Lemma 6.5 (Trace equivalence for labels unification)**.**
Let $(\hat{S}, g)$, $\hat{S} := (\hat{N}, M_{ini}, M_{fin})$, $\hat{N} := (\hat{P}, \hat{T}, \hat{F}, \hat{\Lambda}, \hat{\lambda})$, be a result of sets of labels unification in a net system $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, for $\Delta \subseteq \wp(\mathbb{A})$, and let $\eta \in \hat{\Lambda}^*$. There exists $\rho \in L(S)$ such that $\eta \in expand(\rho, g)$ iff $\eta \in L(\hat{S})$. ⌟

The reader can find a proof of Lemma 6.5 in Appendix A. Note that it trivially holds that $L(S) \subseteq L(\hat{S})$.

For example, according to Lemma 6.5, because there is a label sequence `abcdhi` of the system in Figure 1 such that $\texttt{abcd}\mu\texttt{i} \in expand(\texttt{abcdhi}, \{(\mu, \{\texttt{g}, \texttt{h}\})\})$, it holds that $\texttt{abcd}\mu\texttt{i}$ is a label sequence of the system in Figure 6. Conversely, because $\texttt{abcd}\mu\texttt{fi}$ is a label sequence of the system in Figure 6, there exists a label sequence ρ of the system in Figure 1 such that $\texttt{abcd}\mu\texttt{fi} \in expand(\rho, \{(\mu, \{\texttt{g}, \texttt{h}\})\})$. For instance, label sequence `abcdgfi` justifies the existence of ρ.

Importantly, sets of labels for which sets of labels unification is performed may overlap. For example, Figure 8 shows system $\hat{S}$ that results from a sets of labels unification in net system S shown in Figure 7 for $\{\{\texttt{b}, \texttt{c}\}, \{\texttt{b}, \texttt{d}\}\}$; the fresh places and transitions are highlighted with gray background. Note that both systems are workflow systems. The language of S consists of two strings: `abc` and `ad`, i.e., $L(S) := \{\texttt{abc}, \texttt{ad}\}$

Let $(\hat{S}, g)$ be a result of sets of labels unification in S for $\{\{\texttt{b}, \texttt{c}\}, \{\texttt{b}, \texttt{d}\}\}$. Then, $g = \{(\kappa, \{\texttt{b}, \texttt{c}\}), (\mu, \{\texttt{b}, \texttt{d}\})\}$. It holds that $expand(\texttt{abc}, g) = \{\texttt{abc}, \texttt{a}\kappa\texttt{c}, \texttt{ab}\kappa, \texttt{a}\kappa\kappa, \texttt{a}\mu\texttt{c}, \texttt{a}\mu\kappa\}$ and $expand(\texttt{ad}, g) = \{\texttt{ad}, \texttt{a}\mu\}$. Due to Lemma 6.5, $L(\hat{S}) = expand(\texttt{abc}, g) \cup expand(\texttt{ad}, g)$.

Framing. Intuitively, a result of framing a workflow system is a system that in a special way prepends a fresh transition with a unique label to the initial place of the workflow system and appends a fresh transition with a unique label to the final place of the workflow system.

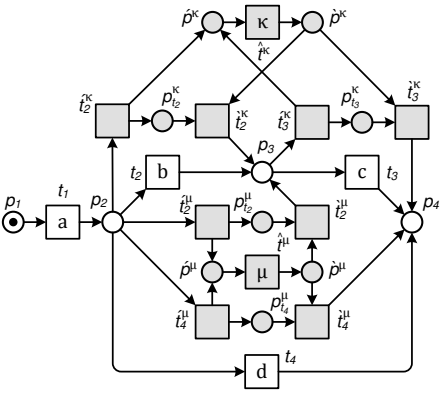


Figure 8: A result of *sets of labels unification* in the net system in Figure 7 for $\{\{\mathtt{b},\mathtt{c}\},\{\mathtt{b},\mathtt{d}\}\}$.

Definition 6.6 (Framing)**.**
A result of *framing* a workflow system $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, with the initial place $i \in P$ and the final place $f \in P$, is a 3-tuple $(\hat{S}, \alpha, \zeta)$, where α and ζ are distinct labels in $\mathbb{A} \setminus \Lambda$, and $\hat{S} = (\hat{N}, \hat{M}_{ini}, \hat{M}_{fin})$, $\hat{N} := (\hat{P}, \hat{T}, \hat{F}, \hat{\Lambda}, \hat{\lambda})$:

- $\hat{P} := P \cup \{p_i, p_f\}$, where $\{p_i, p_f\} \cap P = \emptyset$,
- $\hat{T} := T \cup \{t_i, t_f\}$, where $\{t_i, t_f\} \cap T = \emptyset$,
- $\hat{F} := F \cup \{(p_i, t_i), (t_i, i), (f, t_f), (t_f, p_f)\}$,
- $\hat{\Lambda} := \Lambda \cup \{\alpha, \zeta\}$, $\hat{\lambda} := \lambda \cup \{(t_i, \alpha), (t_f, \zeta)\}$,
- $\hat{M}_{ini} := [p_i]$, and $\hat{M}_{fin} := [p_f]$.

Note that $\hat{P}$, $\hat{T}$, and $\mathbb{A}$ are pairwise disjoint. ⌟

Clearly, framing of a workflow system results in a workflow system.

Proposition 6.7 (Framing)**.**
If (S, α, ζ) is a result of framing a workflow system, then S is a workflow system. ⌟

The proof of Proposition 6.7 follows immediately from the definition of a workflow system and Definition 6.6. Moreover, the language of a system that results from framing a workflow system is in the tight relationship with the language of the workflow system.

Lemma 6.8 (Trace equivalence for framing)**.**
Let $(\hat{S}, \alpha, \zeta)$, $\hat{S} := (\hat{N}, \hat{M}_{ini}, \hat{M}_{fin})$, $\hat{N} := (\hat{P}, \hat{T}, \hat{F}, \hat{\Lambda}, \hat{\lambda})$, be a result of framing a workflow system $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$. Let $\rho \in \Lambda^*$ and $\eta \in \hat{\Lambda}^*$ such that $\eta = \langle\alpha\rangle \circ \rho \circ \langle\zeta\rangle$. Then, $\rho \in L(S)$ iff $\eta \in L(\hat{S})$. ⌟

Again, the proof of Lemma 6.8 follows immediately from Definition 6.6. Figure 9 shows workflow system $\hat{S}$ for which it holds that $(\hat{S}, \alpha, \zeta)$ is a result of framing the workflow system S in Figure 7. Note that because of Lemma 6.8, we know that $L(\hat{S}) = \{\alpha\mathtt{abc}\zeta, \alpha\mathtt{ad}\zeta\}$; recall that $L(S) = \{\mathtt{abc}, \mathtt{ad}\}$.

Sequences test. We say that a label $\alpha \in \mathbb{A}$ is *sole* in a net $N := (P, T, F, \Lambda, \lambda)$ iff $|\{t \in T \mid \lambda(t) = \alpha\}| = 1$. By $tr(N, \beta)$, $\beta \in dom(\lambda)$, we denote a transition in T that has label β, i.e., $tr(N, \beta) := t$, $t \in T$ and $\lambda(t) = \beta$. Note that the value of $tr(N, \beta)$ is unique iff β is a sole label in N. One can use the sequences test insertion, which is defined below,

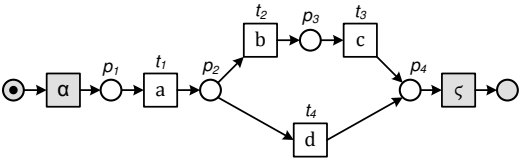


Figure 9: A result of *framing* the workflow system in Figure 7.

to check if a system describes a label sequence with given label subsequences. The definition requires that every label in every given subsequence is sole. Sequences tests are used in Section 6.2 for fresh labels introduced during labels unification and framing transformations; note that by definition these labels are guaranteed to be within the transformed system.

Definition 6.9 (Sequences test)**.**
A result of *sequences test insertion* in a net system $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, for $\Phi \subseteq \mathbb{A}^*$, where every label in every sequence in Φ is sole in N, is a pair $(\hat{S}, g)$, $\hat{S} = (\hat{N}, \hat{M}_{ini}, \hat{M}_{fin})$, $\hat{N} = (\hat{P}, \hat{T}, \hat{F}, \hat{\Lambda}, \hat{\lambda})$, and $g : \mathbb{A} \nrightarrow \Phi$, where:

- g is a bijection such that $\Lambda \cap dom(g) = \varnothing$,
- $\hat{P} := P \cup \{\hat{p}\} \cup \{p_{(\rho,i)} \mid \rho \in \Phi \wedge i \in [1 \,..\, |\rho|)\}$,
- $\hat{T} := T \cup \{t_{(\rho,i)} \mid \rho \in \Phi \wedge i \in [1 \,..\, |\rho|]\}$,
- $\hat{F} := F \cup \{(\hat{p}, t_{(\rho,1)}) \mid \rho \in \Phi\} \cup \{(t_{(\rho,|\rho|)}, \hat{p}) \mid \rho \in \Phi\} \cup$
  $\{(t_{(\rho,i)}, p_{(\rho,i)}) \mid \rho \in \Phi \wedge i \in [1 \,..\, |\rho|)\} \cup$
  $\{(p_{(\rho,i)}, t_{(\rho,i+1)}) \mid \rho \in \Phi \wedge i \in [1 \,..\, |\rho|)\} \cup$
  $\{(p, t_{(\rho,i)}) \mid \rho \in \Phi \wedge p \in \bullet tr(N, \rho_{[i]}) \wedge i \in [1 \,..\, |\rho|]\} \cup$
  $\{(t_{(\rho,i)}, p) \mid \rho \in \Phi \wedge p \in tr(N, \rho_{[i]}) \bullet \wedge i \in [1 \,..\, |\rho|]\} \cup$
  $\{(t, \hat{p}) \mid t \in T \wedge \lambda(t) \neq \tau\} \cup \{(\hat{p}, t) \mid t \in T \wedge \lambda(t) \neq \tau\}$,
- $\hat{\Lambda} := \Lambda \cup dom(g)$, $\hat{\lambda} := \lambda \cup \{(t_{(\rho,i)}, \alpha) \mid g(\alpha) = \rho \wedge i = |\rho|\} \cup \{(t_{(\rho,i)}, \tau) \mid \rho \in \Phi \wedge i \in [1 \,..\, |\rho|)\}$,
- $\hat{M}_{ini} := M_{ini} \uplus [\hat{p}]$, and $\hat{M}_{fin} := M_{fin} \uplus [\hat{p}]$. ⌟

We say that a fresh transition $t_{(\rho,i)}$ *mimics* transition $tr(N, \rho_{[i]})$.

Let A and B be two sets, $\eta \in A^*$, and $g : A \nrightarrow B^*$. Then, $rewrite : A^* \times (A \nrightarrow B^*) \to (A \cup B)^*$ is defined as given below.

$$rewrite(\eta, g) := \begin{cases} \langle\rangle & \eta \text{ is empty} \\ (\eta_{[1]} \in dom(g) \;?\; g(\eta_{[1]}) : \langle \eta_{[1]} \rangle) \circ rewrite(suffix(\eta, 2)) & \text{otherwise} \end{cases}$$

That is, given a string η over an alphabet A and a function g that maps symbols in A to strings over an alphabet B, $rewrite(\rho, g)$ is the string in which every symbol x from the domain of g is replaced with the string $g(x)$. For example, let $\eta := \chi\texttt{cb}\phi\texttt{bcd}\psi\texttt{i}$ and $g := \{(\chi, \alpha\texttt{a}), (\phi, \texttt{de}), (\psi, \mu\texttt{f})\}$. Then, it holds that $rewrite(\eta, g) = \alpha\texttt{acbdebcd}\mu\texttt{fi}$.

Label sequences of a given net system and a result of sequences test insertion in the given system are tightly related.

Lemma 6.10 (Trace equivalence for sequences test)**.**
Let $(\hat{S}, g)$, $\hat{S} := (\hat{N}, \hat{M}_{ini}, \hat{M}_{fin})$, $\hat{N} := (\hat{P}, \hat{T}, \hat{F}, \hat{\Lambda}, \hat{\lambda})$, *be a result of sequences test insertion*

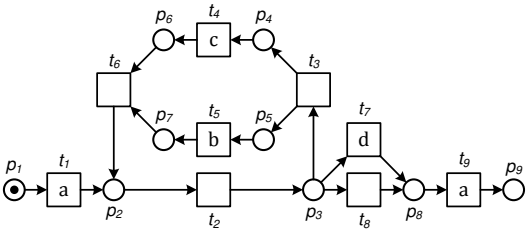


Figure 10: A workflow system.

in a net system $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, *for* $\Phi \subseteq \Lambda^*$. *Let* $\rho \in \Lambda^*$ *and* $\eta \in \hat{\Lambda}^*$ *such that* $\rho = rewrite(\eta, g)$. *Then,* $\rho \in L(S)$ *iff* $\eta \in L(\hat{S})$. ⌟

The proof of Lemma 6.10 is in Appendix A.

Note that a sequence in a set used to perform a sequences test insertion may contain multiple instances of the same element. Also, two different sequences may share same elements. Finally, a net system that is subject to sequences test insertion may contain multiple transitions that have the same label; of course, this label should not be an element of any sequence in the set for which the transformation is performed.

Consider the workflow system $S := (N, [p_1], [p_9])$ in Figure 10.

Figure 11 shows a result $(\hat{S}, g)$, $\hat{S} := (\hat{N}, [p_1, \hat{p}], [p_9, \hat{p}])$, of sequences test insertion in S for $\{\rho, \eta\}$, where $\rho := \texttt{cbc}$, $\eta := \texttt{bd}$, and g is given by the set $\{(\chi, \rho), (\phi, \eta)\}$.

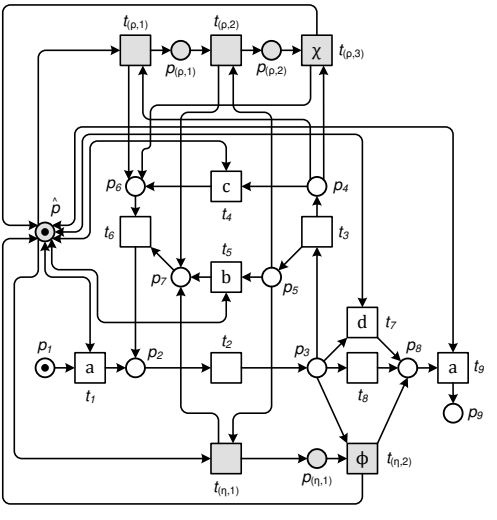


Figure 11: A net system.

For instance, $\texttt{aa} = rewrite(\texttt{aa}, g)$, $\texttt{ada} = rewrite(\texttt{ada}, g)$, $\texttt{abca} = rewrite(\texttt{abca}, g)$, $\texttt{abcda} = rewrite(\texttt{abcda}, g)$, $\texttt{acbcbda} = rewrite(\texttt{a}\chi\phi\texttt{a}, g)$, $\texttt{acbcbda} = rewrite(\texttt{a}\chi\texttt{bda}, g)$, $\texttt{acbcbda} = rewrite(\texttt{acbc}\phi\texttt{a}, g)$, and $\texttt{acbcbbca} = rewrite(\texttt{a}\chi\texttt{bbca}, g)$, etc. Then, according to Lemma 6.10, because $\texttt{acbcbda} \in L(S)$, it holds that $\texttt{a}\chi\phi\texttt{a} \in L(\hat{S})$, and because $\texttt{a}\chi\texttt{bbca} \in L(\hat{S})$, it holds that $\texttt{acbcbbca} \in L(S)$. In fact, the reader can verify that it holds that $\{\texttt{aa}, \texttt{ada}, \texttt{abca}, \texttt{abcda}, \texttt{acbcbda}, \texttt{acbcbbca}\} \subset L(S)$ and $\{\texttt{aa}, \texttt{ada}, \texttt{abca}, \texttt{abcda}, \texttt{a}\chi\phi\texttt{a}, \texttt{a}\chi\texttt{bda}, \texttt{acbc}\phi\texttt{a}, \texttt{a}\chi\texttt{bbca}\} \subset L(\hat{S})$.

6.2. Solving the Trace Executability Problem

This section demonstrates that a workflow system executes a given trace with wildcards iff the language of the net system obtained by performing a sets of labels unification, framing, and sequences test insertion contains a special string. This special string and all the transformations applied on the input system depend on the input trace with wildcards. Let $\eta \in (\mathbb{A} \cup \{*\})^*$. Then, by $maxsubseq(\eta)$ we denote the set of all maximal nonempty sub-sequences of η that do not contain $*$. For example, if $\eta := \langle \mathtt{a}, \mathtt{a}, *, *, \mathtt{a}, \mathtt{a}, *, \mathtt{b}, *, \mathtt{a}, \mathtt{b} \rangle$, then it holds that $maxsubseq(\eta) = \{\langle \mathtt{a}, \mathtt{b} \rangle, \langle \mathtt{a}, \mathtt{a} \rangle, \langle \mathtt{b} \rangle\}$.

Let σ be a sequence, $Set(\sigma)$ denotes the set that contains all the elements of σ.

Theorem 6.11 (Trace executability)**.**
A workflow system $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, executes a trace with wildcards ω iff there exists $\eta \in L(\hat{S})$, $rewrite(\eta|_{dom(h)}, h) = (\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)|_{\mathbb{A}}$, where:

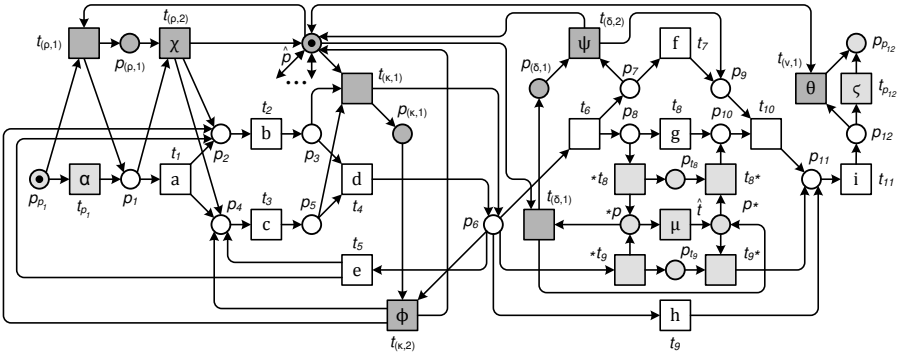
- *$(\hat{S}, h)$ is a result of sequences test insertion in S'' for $maxsubseq(\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)$,*
- *(S'', α, ζ) is a result of framing S',*
- *(S', g) is a result of sets of labels unification in S for $\{X \subseteq \mathbb{A} \mid \exists i \in [1..|\omega|] : X = M_{\mathtt{Event}}(\omega_{[i]})\}$, and*
- *$f(e) := \left\langle g^{-1}(M_{\mathtt{Event}}(e)) \right\rangle$, $e \in Set(\omega) \setminus \{*\}$.* ⌟

The reader can find a proof of Theorem 6.11 in Appendix A.[9]

Given a workflow system S, the corresponding transformed version $\hat{S}$ is obtained in three steps. Let S be the workflow system in Figure 1. Then, Figure 12 shows its corresponding transformed version $\hat{S}$ for the trace with wildcards $\omega := \langle (\mathtt{a}, 1.0), *, (\mathtt{d}, 1.0), (\mathtt{e}, 1.0), *, (\mathtt{j}, 0.75), (\mathtt{f}, 1.0), * \rangle$, which can be seen as the meaning of the `Trace` construct of PQL with a concrete encoding '`<a,*,d,e,*,~j,f,*>`'; we assume that '`~`' defaults to the value of 0.75. Note that most of the labels used in the trace are the short names for the full activity labels listed in the caption of Figure 1, while the fresh short name `j` stands for the full activity label of "archive booking request". Note also that fresh place $\hat{p}$ must be connected by double arcs with every observable transition of $\hat{S}$; not explicitly shown in the figure. We further assume that $M_{\mathtt{Event}}(\omega_{[1]}) = \{\mathtt{a}\}$, $M_{\mathtt{Event}}(\omega_{[3]}) = \{\mathtt{d}\}$, $M_{\mathtt{Event}}(\omega_{[4]}) = \{\mathtt{e}\}$, $M_{\mathtt{Event}}(\omega_{[6]}) = \{\mathtt{g}, \mathtt{h}\}$, and $M_{\mathtt{Event}}(\omega_{[7]}) = \{\mathtt{f}\}$ and, hence, it holds that $L(\omega) = \mathtt{a} \circ (\mathbb{A}^*) \circ \mathtt{d} \circ \mathtt{e} \circ (\mathbb{A}^*) \circ (\mathtt{g} \cup \mathtt{h}) \circ \mathtt{f} \circ (\mathbb{A}^*)$.

In the first step, a result of sets of labels unification in workflow system S for $\{\{\mathtt{a}\}, \{\mathtt{d}\}, \{\mathtt{e}\}, \{\mathtt{g}, \mathtt{h}\}, \{\mathtt{f}\}\}$ is constructed. For example, the pair (S', g), where S' is shown in Figure 6 and $g := \{(\mathtt{a}, \{\mathtt{a}\}), (\mathtt{d}, \{\mathtt{d}\}), (\mathtt{e}, \{\mathtt{e}\}), (\mu, \{\mathtt{g}, \mathtt{h}\}), (\mathtt{f}, \{\mathtt{f}\})\}$ is one possible result of this unification. In the second step, a result (S'', α, ζ) of framing S' is obtained, while in the third step a result of sequences test insertion in S'' for $\{\langle \alpha, \mathtt{a} \rangle, \langle \mathtt{d}, \mathtt{e} \rangle, \langle \mu, \mathtt{f} \rangle, \langle \zeta \rangle\}$ is constructed. Note that S' and S'' are workflow systems because of Proposition 6.2 and Proposition 6.7, respectively. For example, the pair $(\hat{S}, h)$, where $\hat{S}$ is the system in Figure 12 and $h := \{(\chi, \langle \alpha, \mathtt{a} \rangle), (\phi, \langle \mathtt{d}, \mathtt{e} \rangle), (\psi, \langle \mu, \mathtt{f} \rangle), (\theta, \langle \zeta \rangle)\}$ is one possible result of performing second and third transformation steps; in the figure we denote $\langle \alpha, \mathtt{a} \rangle$, $\langle \mathtt{d}, \mathtt{e} \rangle$, $\langle \mu, \mathtt{f} \rangle$, and $\langle \zeta \rangle$, by ρ, κ, δ, and ν, respectively. The

[9] Recall that function $M_{\mathtt{Event}}$ is defined in Section 4.1.



$\gamma :=$

$\gg$	χ	$\gg$	$\gg$	$\gg$	ϕ	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	ψ	$\gg$	$\gg$	θ
τ	χ	b	c	τ	ϕ	b	c	d	τ	τ	τ	τ	ψ	τ	i	θ
$t_{(\rho,1)}$	$t_{(\rho,2)}$	t_2	t_3	$t_{(\kappa,1)}$	$t_{(\kappa,2)}$	t_2	t_3	t_4	t_6	$*t_8$	$t_{(\delta,1)}$	t_8*	$t_{(\delta,2)}$	t_{10}	t_{11}	$t_{(v,1)}$

Figure 12: A net system $\hat{S}$ obtained after performing sets of labels unification, framing, and sequences test insertion in the workflow system in Figure 1 in order to compute `Executes(<a,*,d,e,*,~j,f,*>)`, where `j` := "archive booking request", and an optimal alignment between $\langle \chi, \phi, \psi, \theta \rangle$ and $\hat{S}$.

fresh places and transitions introduced during the second and third steps of the transformation are highlighted with dark gray background in the figure. It trivially holds that $\{\langle \alpha, \mathtt{a} \rangle, \langle \mathtt{d}, \mathtt{e} \rangle, \langle \mu, \mathtt{f} \rangle, \langle \zeta \rangle\} = maxsubseq(\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)$, where f is the set $\{((\mathtt{a}, 1.0), \mathtt{a}), ((\mathtt{d}, 1.0), \mathtt{d}), ((\mathtt{e}, 1.0), \mathtt{e}), ((\mathtt{j}, 0.75), \mu), ((\mathtt{f}, 1.0), \mathtt{f})\}$ and, therefore, $\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle = \langle \alpha, \mathtt{a}, *, \mathtt{d}, \mathtt{e}, *, \mu, \mathtt{f}, *, \zeta \rangle$.

According to Theorem 6.11, one can check whether the workflow system in Figure 1 executes a trace with wildcards encoded as '`<a,*,d,e,*,~j,f,*>`' by checking if the language of the net system in Figure 12 contains a string η such that $rewrite(\eta|_{dom(h)}, h) = \langle \alpha, \mathtt{a}, \mathtt{d}, \mathtt{e}, \mu, \mathtt{f}, \zeta \rangle$. One can verify if that is the case using the result of Lemma 6.12.

The *move on trace* cost function over a net system S is the function $c : \mathbb{M}_S \to \{0, 1\}$ such that $c((x, y)) := 1$ if $x \in \mathbb{A}$ and $y = \gg$; otherwise $c((x, y)) := 0$, $(x, y) \in \mathbb{M}_S$.

Lemma 6.12 (Trace executability)**.**
There exists $\eta \in L(\hat{S})$ such that $rewrite(\eta|_{dom(h)}, h) = (\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)|_{\mathbb{A}}$ iff $c(\gamma) = 0$, where:

- *$(\hat{S}, h)$ is a result of sequences test insertion in S'' for $maxsubseq(\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)$,*
- *(S'', α, ζ) is a result of framing S',*
- *(S', g) is a result of sets of labels unification in S for $\{X \subseteq \mathbb{A} \mid \exists\, i \in [1..|\omega|] : X = M_{\mathtt{Event}}(\omega_{[i]})\}$,*
- *$f(e) := \left\langle g^{-1}(M_{\mathtt{Event}}(e)) \right\rangle$, $e \in Set(\omega) \setminus \{*\}$,*
- *S is a workflow system,*
- *ω is a trace with wildcards,*
- *γ is an optimal alignment between ρ and $\hat{S}$,*
- *ρ is a finite sequence of symbols over $dom(h)$ such that $rewrite(\rho, h) = (\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)|_{\mathbb{A}}$, and*
- *c is the move on trace cost function over $\hat{S}$.*

Refer to Definition 3.7 for the definition of an optimal alignment between a trace and

$\gamma_1 :=$

α	a	$\gg$	$\gg$	d	e	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	μ	$\gg$	f	$\gg$	$\gg$	ζ
α	a	b	c	d	e	b	c	d	τ	τ	μ	τ	f	τ	i	ζ
t_{p_1}	t_1	t_2	t_3	t_4	t_5	t_2	t_3	t_4	t_6	$*t_8$	$\hat{t}$	t_8*	t_7	t_{10}	t_{11}	$t_{p_{12}}$

$\gamma_2 :=$

a	$\gg$	$\gg$	d	e	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	μ	$\gg$	f	$\gg$	$\gg$
a	b	c	d	e	b	c	d	τ	τ	μ	τ	f	τ	i
t_1	t_2	t_3	t_4	t_5	t_2	t_3	t_4	t_6	$*t_8$	$\hat{t}$	t_8*	t_7	t_{10}	t_{11}

$\gamma_3 :=$

a	$\gg$	$\gg$	d	e	$\gg$	$\gg$	$\gg$	$\gg$	g	f	$\gg$	$\gg$
a	b	c	d	e	b	c	d	τ	g	f	τ	i
t_1	t_2	t_3	t_4	t_5	t_2	t_3	t_4	t_6	t_8	t_7	t_{10}	t_{11}

Figure 13: Alignments between sample traces and systems in Figs. 1 and 12.

system. The reader can find a proof of Lemma 6.12 in Appendix A.

Hence, it holds that the language of the net system $\hat{S}$ in Figure 12 contains a string η such that $rewrite(\eta|_{dom(h)}, h) = \langle \alpha, \mathtt{a}, \mathtt{d}, \mathtt{e}, \mu, \mathtt{f}, \zeta \rangle$ iff the cost of an optimal alignment between $\langle \chi, \phi, \psi, \theta \rangle$ and $\hat{S}$ as per the move on trace cost function c over $\hat{S}$ is equal to zero; this holds due to the fact that $rewrite(\langle \chi, \phi, \psi, \theta \rangle, h) = \langle \alpha, \mathtt{a}, \mathtt{d}, \mathtt{e}, \mu, \mathtt{f}, \zeta \rangle$. Note that c defines costs of moves in $\{(x, \gg) \mid x \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \alpha, \zeta, \mu, \chi, \phi, \psi, \theta\}\}$ to be equal to one, while costs of all the other moves to be equal to zero. Figure 12 also demonstrates an optimal alignment γ between $\langle \chi, \phi, \psi, \theta \rangle$ and $\hat{S}$ for which it holds that $c(\gamma) = 0$; indeed, γ does not contain a move on trace. Therefore, according to Lemma 6.12, there exists a string $\eta \in L(\hat{S})$ such that $rewrite(\eta|_{dom(h)}, h) = (\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)|_{\not\Lambda}$, for example $\eta := \chi \mathtt{bc} \phi \mathtt{bcd} \psi \mathtt{i} \theta$ is such a string which is a label sequence of $\hat{S}$ induced by the execution which is used in alignment γ in the figure and, thus, according to Theorem 6.11, the workflow system in Figure 1 executes the trace with wildcards encoded in '<a,*,d,e,*,~j,f,*>'.

The last result of this subsection may sometimes help to avoid the above-proposed complex computations.

Proposition 6.13 (Trace executability)**.**
Let $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, be a net system and let ω be a trace with wildcards. If there exists an element $e \in \not\Lambda \times [0, 1]$ of ω such that $M_{\mathtt{Event}}(e) \cap \Lambda = \emptyset$, then S does not execute ω. ⌟

This result follows from the notion of the language of a net system and Definition 4.4. It was used in the evaluation of scenario-based process querying reported in Section 7.

6.3. Interpreting Results

An optimal alignment that is used to justify that a given workflow system executes a trace with wildcards, refer to Lemma 6.12, can also be used to explain why the system executes the trace. This optimal alignment can be transformed into a fresh alignment that demonstrates how observable transitions of the system can be executed according to the order specified by the trace. We perform the transformation in three steps. Intuitively, these steps 'compensate' the effects of the sets of labels unification, framing, and sequences test insertion on the optimal alignment. For example, optimal alignment γ in Figure 12 used to justify that the workflow system in Figure 1 executes the trace with wildcards '<a,*,d,e,*,~j,f,*>' can be transformed into alignment γ_3 in Figure 13

between the trace $\langle \mathtt{a},\mathtt{d},\mathtt{e},\mathtt{g},\mathtt{f}\rangle$ and the execution $\langle t_1,t_2,t_3,t_4,t_5,t_2,t_3,t_4,t_6,t_8,t_7,t_{10},t_{11}\rangle$ of the system in Figure 1.

In the first step, alignment γ_1 between $\mathit{rewrite}(\langle \chi,\phi,\psi,\theta\rangle,h) = \langle \alpha,\mathtt{a},\mathtt{d},\mathtt{e},\mu,\mathtt{f},\zeta\rangle$ and system $\hat{S}$ in Figure 12 is constructed, refer to Figure 13. It is obtained by replacing moves on system and synchronous moves in γ that are defined for the fresh transitions introduced during the sequences test insertion with synchronous moves defined for transitions that are mimicked by the corresponding fresh transitions (recall from Section 6.2 that $h := \{(\chi,\langle\alpha,\mathtt{a}\rangle),(\phi,\langle\mathtt{d},\mathtt{e}\rangle),(\psi,\langle\mu,\mathtt{f}\rangle),(\theta,\langle\zeta\rangle)\}$); that is, moves $(\gg,t_{(\rho,1)})$, $(\chi,t_{(\rho,2)})$, $(\gg,t_{(\kappa,1)})$, $(\phi,t_{(\kappa,2)})$, $(\gg,t_{(\delta,1)})$, $(\psi,t_{(\delta,2)})$, and $(\theta,t_{(\nu,1)})$ in γ get replaced with moves (α,t_{p_1}), $(\mathtt{a},t_1)$, $(\mathtt{d},t_4)$, $(\mathtt{e},t_5)$, $(\mu,\hat{t})$, $(\mathtt{f},t_7)$, and $(\zeta,t_{p_{12}})$ in γ_1, respectively. To compensate the effects of framing, alignment γ_2 in Figure 13 is constructed by removing the first and the last move in alignment γ_1. Finally, alignment γ_3 is obtained from γ_2 by removing all the moves on system for the silent transitions introduced during the sets of labels unification (transitions $*t_8$ and t_8* in the running example) and replacing the synchronous moves defined for solitary transitions introduced during labels unifications with the synchronous moves for the corresponding 'unified' transitions of the original system ($(\mu,\hat{t})$ is replaced with $(\mathtt{g},t_8)$ in the running example).

The sequence of transitions used to define alignment γ_3 in Figure 13, i.e., $\langle t_1,t_2,t_3,t_4,$ $t_5,t_2,t_3,t_4,t_6,t_8,t_7,t_{10},t_{11}\rangle$, is an execution of the system in Figure 1 which witnesses that the system indeed executes '`<a,*,d,e,*,~j,f,*>`'. This execution induces the label sequence `abcdebcdgfi` which is in the language of the given trace with wildcards; it starts with `a` and contains substring `de` which is eventually followed by substring `gf`. Note that in this example, short activity name `g` is accepted as a valid substitution of '`~j`'; recall that in this running example we assume that '~' defaults to the label similarity threshold of 0.75 and $M_{\mathtt{Event}}((\mathtt{j},0.75)) = \{\mathtt{g},\mathtt{h}\}$. Tools that implement scenario-based process querying method can use alignments obtained via the above-described procedure to highlight and/or replay executions that demonstrate why the corresponding `Executes` predicates evaluate to *true* for the retrieved workflow systems.

7. Evaluation

The proposed querying approach has been implemented and is publicly available under the open source GNU Lesser General Public License.[10] The implementation exhibits a well-defined application programming interface (API) to facilitate its integration with other software products. The PQL tools is integrated into the Apromore process model repository [35]. The PQL tools can also be used from the command line.

The implementation supports multi-threaded querying. To avoid unnecessary computations when evaluating `Executes` predicates, the tool implements the label-based filtering that relies on the result captured in Proposition 6.13. That is, given a trace with wildcards, for each of its elements, a net system must contain an observable transition with a label contained in the set that stems from the interpretation of that element; otherwise the system does not execute the given trace. The tool uses an ANTLR [68]

[10] `https://github.com/processquerying/PQL.git`

generated parser that can build and walk syntax trees of PQL queries and the implementation for computing optimal alignments available via the ProM framework [69]. The tool can be configured to use one of the three integrated information retrieval engines for scoring label similarities. These are Apache Lucene[11], Themis-IR [70], and an implementation of the label similarity scoring approach based on the Levenshtein distance. Note that the Apache Lucene and Themis-IR engines are configured to use the vector space model to perform label similarity assessments. The Levenshtein distance approach for label similarity is implemented based on the principles proposed in [71]. Note that all the experiments were conducted using the Apache Lucene engine.

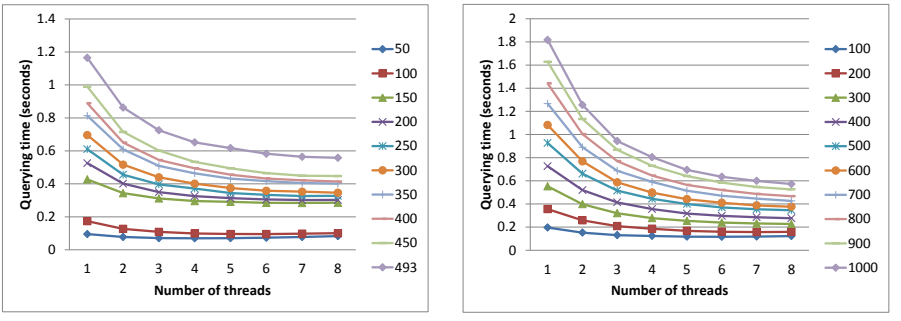
Using this implementation, we conducted three experiments to assess performance of the tools in terms of run times of PQL queries. We assess the impact of query size, number of computation threads, and characteristics of process models on the run times. The experiments were performed on a 6 Core Intel Xeon CPU 3.5 GHz computer with enabled virtualization (12 logical cores), 128GB of RAM, running Windows Server 2012 R2 SE and Java VM 1.8 (with standard allocation of memory).

Datasets. In the experiments, we used 493 industrial and 1,000 synthetic sound workflow systems. The industrial workflow systems were obtained from the SAP R/3 Reference Model [3], a collection of 604 EPCs from different domains. The EPCs were converted to Petri net systems and subsequently completed to workflow systems, using the techniques proposed in [72, 73]. The unsound workflow systems were filtered out, resulting in a collection of 493 sound workflow systems which was used in the experiments. The synthetic collection of workflow systems was generated using the tool described in [74]. The tool takes as input a seed collection and creates process models with similar structural and label characteristics to models in the seed collection. We used the SAP R/3 collection as a seed and generated 30,200 synthetic EPCs. The EPCs were converted to workflow systems and the unsound systems were filtered out. We then randomly selected 1,000 sound workflow systems which were used in the experiments. The systems in the industrial and the synthetic collections have similar structural characteristics in terms of the average numbers of transitions (15.91 and 13.52 for the industrial and the synthetic models, respectively), observable transitions (8.57 and 10.95), places (17.85 and 16.63), flow arcs (35.7 and 32.4), places with multiple output transitions (0.64 and 0.45), places with multiple input transitions (0.65 and 0.43), transitions with multiple output places (1.13 and 1.26), and transitions with multiple input places (1.13 and 1.22).[12]

In the experiments, randomly generated traces with wildcards were used. To generate a trace, we performed the following procedure: First, a random net system was selected from the collection. Second, a random execution of the selected system was generated by firing random enabled transitions. Third, we created a sequence of labels (of the required length) from the labels of the observable transitions of the generated execution starting from a random position in the execution (the order of labels in the sequence follows the order of corresponding transitions). Fourth, we replaced a random number

[11] `https://lucene.apache.org/`

[12] The synthetic collection is available at:
`https://github.com/processquerying/PQL/tree/master/pnml`.



(a) (b)

Figure 14: The average querying times for different numbers of query threads and (a) industrial and (b) synthetic models.

of labels in the obtained sequence of labels with the special wildcard character '*' (one or two replacements were applied). Finally, we inserted the tilde grapheme '~' before a random number of labels in the sequence (the number of inserted *tilde* symbols ranged from zero to two). If we were not able to create a trace of a given length from the execution generated during the second step, e.g., the execution was too short, then the procedure was repeated from the beginning. The above-outlined trace generation procedure was used in all experiments.

Experiment 1: Impact of trace length on querying time. In the first experiment, we evaluated the impact of the trace length on querying times. We varied the trace length from four to ten and generated 1,000 random traces with wildcards of each length. The querying was performed over all the models in each collection, i.e., industrial and synthetic, using one query thread. To eliminate load times, we repeated each test four times and recorded average times of the second to fourth repetitions.[13]

We have observed that the average run times range from 0.9 seconds (for traces of length four) to 1.19 seconds (for traces of length ten) for the industrial collection, and from 1.75 seconds (for traces of length four) to 2.01 seconds (for traces of length ten) for the synthetic collection. On average, each query retrieved 0.27 process models in the industrial collection and 0.15 in the synthetic collection. For the industrial data set, the minimum recorded query time is 0.61 seconds (for a trace of length four), and the maximum query time is 14.3 s (for a trace of length nine). For the synthetic data set, the minimum query time was 1.41 seconds (for a trace of length four), and the maximum query time was 10.55 seconds (for a trace of length eight).

For both model collections, the observed relation between the querying time and the trace length is best described by polynomial functions. These functions are $y = 0.0036 \times x^2 - 0.005 \times x + 0.8716$ (coefficient of determination $R^2 = 0.8917$) and $y = 0.0026 \times x^2 + 0.006 \times x + 1.6879$ ($R^2 = 0.9247$) for the industrial and synthetic collection, respectively. Note that one also obtains good fits with linear approximations. In particular, for the industrial collection, the relation between the querying time and the length of a trace can be captured by $y = 0.0448 \times x + 0.7114$ ($R^2 = 0.8752$), whereas for

[13] This approach to measuring query run times was used in all experiments

the synthetic collection by $y = 0.043 \times x + 1.5691$ ($R^2 = 0.9143$).

Experiment 2: Impact of query threads on querying time. In the second experiment, we varied the number of query threads (from one to eight) and the number of queried process models (with an increment of 50 models for the industrial collection and an increment of 100 models for the synthetic collection). Queries were performed on different combinations of numbers of process models and numbers of query threads for 1,000 random traces with lengths ranging from four to ten. The resulting average querying times are depicted in Figure 14(a) and Figure 14(b) for the industrial and the synthetic models, respectively. They demonstrate that with the increase of the number of query threads querying times decrease, though performance gains get less pronounced when we add more and more computation threads.

For example, querying over all the models in the synthetic collection was accomplished in 1.82 s with one thread, in 1.26 s with two threads (1.44 times faster than with one thread), and 0.57 s with eight threads (3.19 times faster than with one thread). This relation is best captured by the power function $y = 1.8109 \times x^{-0.575}$ ($R^2 = 0.9955$), where y is the querying time and x is the number of query threads. The relation between the querying time and the number of threads over all the models in the industrial collection is captured by $y = 1.1179 \times x^{-0.36}$ ($R^2 = 0.9819$). Note that similar trends were observed for all other experiments over different numbers of process models for both collections.

This experiment has revealed a linear relation between querying time and the size of a process model collection. For example, when querying with one thread was performed over process models from the industrial collection, the average querying time was 0.17 s for 100 models, 0.53 s for 200 models, 0.70 s for 300 models, 0.89 s for 400 models, and 1.16 s for all 493 models. The relation on all such observations is best described by the linear function $y = 0.1127 \times x + 0.0188$ ($R^2 = 0.9828$), where y is the querying time and x is the number of process models in the collection. A similar trend was observed for different numbers of query threads for both collections.

Experiment 3: Querying times for individual models. In the third experiment, we measured run times of PQL queries on individual models for 1,000 random traces with wildcards of lengths ranging from four to ten. This experiment was performed on all the models in both collections using one query thread. During querying over the industrial models, which consisted of executing 1,000 queries over 493 models, in 491,347 cases process models were filtered using our label-based filtering approach (we refer to them here as *filtered* cases), while our alignment-based approach was applied in the remaining 1,653 cases (we refer here to the corresponding cases as *aligned* cases). During querying on the synthetic models (1,000 queries over 1,000 models), in 999,000 cases process models were filtered and in 1,000 cases the alignment-based approach was applied. For 99.84% of filtered cases in the industrial collection and for 99.73% in the synthetic collection querying times were less than 2 ms per model; while for 0.004% of filtered cases in the industrial collection and for 0.003% in the synthetic collection they were more than 50 ms per model. For 69.4% of aligned cases in the industrial data set and for 71.6% of aligned cases in the synthetic data set querying times were less than 20 ms per model, while for 4.3% in the industrial data set and for 9.8% in the synthetic data set they were more than 500 ms. The average querying times for filtered cases were 1.56 ms and 1.66 ms for the industrial and synthetic models, respectively; while for aligned cases

they were 194.41 ms for the industrial models and 248.05 ms for the synthetic models. Overall, in the industrial collection, the average query run time per single process model was 2.2 ms, with the minimum time of 0.6 ms and the maximum time of 12.2 seconds, while in the synthetic collection, the average query time per model was 1.91 ms, with the minimum time of 0.7 ms and the maximum time of 8.3 seconds.

The querying times for the non-filtered cases are best explained by state spaces of the corresponding process models. The relation between the querying times and the sizes of workflow system state spaces is captured by the power function $y = 0.6992 \times x^{0.6699}$ ($R^2 = 0.9297$) for the industrial collection, and $y = 0.6817 \times x^{0.6967}$ ($R^2 = 0.9257$) for the synthetic collection. One workflow system in the industrial collection has a much bigger state space (more than two millions reachable states, as measured by the LoLA model checker [75]) compared to state spaces of the other models in the collection (less than 300,000 reachable states). The (16) longest querying times (ranging from 4.7 seconds to 12.2 seconds) for the aligned cases in this collection were recorded for this system.

The conducted experiments clearly demonstrate the feasibility of applying the proposed querying technique in practical settings. Being close to real time is an important characteristic of our process querying approach as business analysts often need to perform many exploratory queries when working on practical tasks.

8. Expressiveness, Computability, and Complexity

This section complements the above discussions by looking at the expressiveness, computability, and complexity of the scenario-based process querying problem. Although the main results of the scenario-based querying were devised for workflow systems, refer to Section 6, the most computationally demanding step, i.e., computation of alignments, is performed on the net systems resulting from the transformed versions of workflow systems Hence, the subsequent discussions of this section revert to the analysis of the general formulation of the problem over net systems.

8.1. Expressiveness

The expressiveness of a programming language refers to the variety of statements that the language can capture. Intuitively, if every statement in language A can also be captured in language B, but some statements in language B cannot be captured in language A, then language B is more expressive than language A [76].

Scenario-based querying increases the expressiveness of PQL. PQL without the support of scenario-based querying can express an intent to retrieve models that satisfy a condition over a finite repertoire of behavioral predicates of the 4C spectrum [67]. The expressiveness of a query language that relies on the use of a finite collection of behavioral predicates has fundamental limitations, as one can only distinguish between a *finite* number of model classes [77], where for every model from the same class the predicates evaluate to the same values. Scenario-based querying can be used to express an intent to retrieve models that describe a language that contains (or does not contain) *any* finite language to discriminate *infinitely* many models.

Scenario-based querying is less expressive than temporal logics. For example, one can check whether a model describes a trace with wildcards using CTL, as shown below.

However, it is not known whether temporal logics are more expressive than the full PQL. As of today, several problems on model checking the properties of the 4C spectrum, i.e., the core primitives of PQL, are open [67, 78].

The trace executability problem that underpins scenario-based querying method can be tackled using general purpose *model checking* techniques [37]. Given a model of a finite-state system and a formal property, model checking techniques check whether the property holds for the system. Formal properties (to be checked by the model checking techniques) are usually specified using *temporal logics*. Temporal logics are often classified into two groups: linear time and branching time logics. Some properties can be captured more naturally in a branching time logic than in linear time logic, and vice-versa. Branching time logics are better suited to capture the trace executability problem. For net systems, properties specified in linear time logics are usually interpreted over all maximal occurrence sequences, including those that do not reach the final marking, making it difficult to express a property about *some* execution of the system.

Branching time logics over net systems are usually interpreted over the corresponding reachability graphs. One can check whether a system describes an execution that visits markings that cover each marking in a sequence of markings $\langle M_1, M_2, \ldots, M_n \rangle$, $n \in \mathbb{N}$, in the order specified by the sequence, using CTL, a widely-used branching time logic, as follows:

$$\exists \Diamond (cover(M_1) \wedge (\exists \Diamond (cover(M_2) \wedge (\ldots (\exists \Diamond cover(M_n)) \ldots)))).$$

In this formula template, $\exists\Diamond\Phi$, Φ is a CTL state formula quantified using the temporal modality "eventually", i.e., $\exists\Diamond\Phi \equiv \exists(true \bigcup \Phi)$; $\bigcup$ is the "until" temporal modality, refer to [37] for more info. The $cover(M)$ predicate, where M is a marking, is the basic predicate which checks if M is covered at a current marking M'. That is, $cover(M)$ evaluates to *true* iff for every place p it holds that $M(p) \leq M'(p)$. The $cover(M)$ predicate is usually captured as the conjunction $\bigwedge_{p \in M}(ge(p, M(p)))$, where the $ge(p,x)$ predicate checks if the number of tokens at place p is greater than or equal to x, i.e., $ge(p,x)$ evaluates to true at a marking M' iff $M'(p) \geq x$.

One can use the above proposed CTL formula template to specify the *trace executability* problem (Definition 4.4). In general, the fact that one can observe a sequence of markings in an execution of a net system does not guarantee that one observes a certain sequence of transition occurrences. Hence, the above CTL formula must be verified on the transformed version of the net system. Similar to the sequences test transformation (Definition 6.9), this transformation must enforce a correspondence between observations of markings and transition occurrences. After the sequences test transformation is applied, once the preset of an observable transition t introduced during the transformation is marked it is enforced that t is the next transition to occur. For example, coming back to the motivating example R3 from 2.3, which is also detailed in Section 6, one can express the problem of checking whether the net system in Figure 1 executes the trace with wildcards `<a,*,d,e,*,~j,f,*>` using the CTL formula that checks whether the net system in Figure 12 describes an execution that first reaches a marking that covers $[p_{(\rho,1)}]$, then a marking that covers $[p_{(\kappa,1)}]$, then a marking that covers $[p_{(\delta,1)}]$, and eventually reaches $[p_{p_{12}}, \hat{p}]$. Unfortunately, as discussed in Section 8.2, even if one succeeds in capturing the trace executability problem in CTL,

(or LTL, which is the most widely-used linear time logic), the problem of checking properties captured using CTL, or LTL, is in general undecidable for net systems. Given a class of formulas that can express the trace executability problem in some temporal logic, one can study whether this class of formulas is computable. However, as of today, the precise definition of such a class and decidability of its formulas are open problems.

8.2. Computability and Complexity

The computability of a problem refers to the ability to solve the problem using algorithms. Our scenario-based process querying method is computable. Given a net system S (Definition 3.3) and a trace with wildcards ω (Definition 4.1), one can compute if S executes ω, i.e., the problem of *trace executability* (Definition 4.4) is decidable. First, to check trace executability, S is transformed using the sets of labels unification (Definition 6.3), framing (Definition 6.6), and sequences test (Definition 6.9) transformations. The size of the resulting transformed net system, i.e., the number of places and transitions, uses $O(|S|+|\omega|)$ space, where $|S|$ and $|\omega|$ are the size of S and the length of ω, respectively. Next, an optimal alignment between the rewritten version of ω and the transformed net system is computed (Lemma 6.12). The size of the rewritten trace uses $O(|\omega|)$ space, refer to Section 6.2. The problem of computing an optimal alignment between a trace and net system is equivalent to the reachability problem [58], which is decidable [79] with the exponential space as the best-known lower bound [80].

Properties of net systems expressed using temporal logics are mostly undecidable. For example, the model checking problem for net systems and LTL, or CTL, is undecidable [38]. Note that the model checking problem is also undecidable for net systems and UB^{-}, which is one of the weakest known branching time logics [38]. This particular logic, however, includes predicates and operators of CTL that are used to express the trace executability problem in Section 8.1.

Despite its high computational complexity, the proposed scenario-based process querying method works in (close to) real time on industrial and synthetic net systems, refer to Section 7. The fact allows using the approach in practice, which includes those common situations when no correctness criteria on net systems are imposed [4].

We implemented the technique that captures the trace executability problem in CTL (as proposed in Section 8.1) and then solves it using the LoLA model checker [75], one of the state-of-the-art model checkers offered by the academic community. We observed that the execution times of this implementation are unpredictable, which is consistent with the undecidability result for model checking over net systems. For example, most of our attempts to model check the trace executability problem of the motivating example R3 from Section 2.3, which is also discussed in detail in Section 6, did not terminate within ten minutes. Note that our implementation of the querying method proposed in this paper has repeatedly (for all the attempts we performed) solved this very same instance of the problem under one second. These observations can be explained by the fact that LoLA is a general purpose model checker that applies heuristics to traverse a possibly infinite number of reachable states of the system and, thus, may fail to discover a solution to the problem, whereas the alignment-based technique is tailored to solve the trace executability problem and exploits its specifics, like the reachability of the final marking of the system.

9. Related Work

In general, querying deals with retrieving information that is relevant to a given *information need* from a collection of *information resources*. In process querying, information resources are repositories of process models. Information needs that can be satisfied using the process querying technique proposed in this paper address information about possible scenarios, or instances, described in process models.

In [33, 81], we performed a systematic literature review of the state-of-the-art methods for querying repositories of process models. In that work, we developed a framework for classifying process querying methods. According to our framework, scenario-based process querying addresses querying of *formal* process models using a query language with a *formal* semantics that implements the *read* querying intent, i.e., is designed to retrieve models from repositories. As such, it addresses the major gap in the area of process querying identified in [34, 33].

Next, we discuss *model checking* and *graph database querying*, which are two established approaches that can be used to implement process querying, and the state-of-the-art dedicated techniques for querying repositories of process models based on structural and behavioral properties of the models. For behavioral querying methods, we differentiate between techniques that rely on behavioral abstractions and methods that perform queries over exact instances encoded in the models.

Model checking studies problems that can verify various properties of process models [82, 37]. A model checking problem is a problem that, given a formal specification of a property, usually captured using some *specification language*, and a process model, answers whether the property holds in the given model. Often, to solve a given problem, a model checking technique proceeds by constructing an alternative representation of the model that indicates whether the property of interest holds in the model. Model checking techniques usually use *temporal logics* as property specification languages, e.g., linear temporal logic (LTL) and computational tree logic (CTL). Model checking techniques can be used to retrieve process models that fulfill a property of interest [83]. Such a property, for example, can specify a request to check whether a process model describes a scenario in which activity labels are arranged in a certain order.

Similar to model checking, scenario-based querying makes decisions based on alternative representations of process models, i.e., transformed net systems, like the one shown in Figure 12. These representations may capture infinite-state systems; recall that the proposed querying technique operates on easy-sound systems, which, in general, can be *unbounded* [84], i.e., can describe an infinite number of reachable states/markings. Note, however, that model checking techniques are subject to decidability issues; for infinite-state systems model checking is in general not effectively computable [37]; refer to the discussion in Section 8. Moreover, temporal logics, such as LTL and CTL, require profound expert knowledge to specify a property to be verified, while PQL adopts an intuitive syntax for specifying templates of process scenarios. Finally, a model checking problem is a *decision problem* with a yes-or-no answer, i.e., 'yes' if the property holds and 'no' otherwise. If a property is violated in a model, model checking techniques can generate a counterexample that indicates how the model could reach the undesired state [37]. In contrast, the proposed process querying approach generates an execution that justifies that the requested property indeed holds in the retrieved model, refer to

Section 6.3 for details.

Process models are often captured as directed graphs. Thus, a collection of process models can be seen as a graph database [85]. Consequently, one may use graph querying techniques to realize process querying [86], similar to that proposed in this paper. In the area of graph databases, there are various techniques to support graph querying, such as regular path queries, graph pattern matching, and graph similarity techniques [87, 86, 85, 88, 89]. However, process models are special graphs in which vertices may have different types that encode different control flow logic. As a result, one cannot reduce the problem of checking whether a process model captures an execution in which activities are observed in a certain order to a problem of checking the existence of a structural pattern in the model; this phenomenon is demonstrated, for instance, with query Q_1 and model 4 in Section 5.4. One may attempt to check whether a system executes a trace with wildcards by querying the graph that encodes its state space, i.e., the graph of all reachable states and state transitions described in the model of the system. However, these graphs can be immense in size, even for small systems [90]. For example, the state space of one workflow system with 91 vertices (places and transitions) and 110 arcs from the evaluation discussed in Section 7 contains 2,097,422 vertices (states) and 22,021,078 edges (state transitions)! From bad to worse, a state space of an unbounded process model is infinite and, hence, cannot be stored and processed on a computer, whereas our technique can work with process models that specify infinite state spaces.

BP-QL [91] and BPMN-Q [92] are two typical examples of query languages that can be used to express intents to retrieve process models based on paths and structural patterns in their underlying graphs. These languages can be seen as adaptations of graph query languages to the specifics of graphs that are used to compose process models. In addition to exact structural matching, structural similarity search is also used to retrieve process models. In [93], the authors use graph isomorphism and graph-edit distance techniques to retrieve process models that score an exact match or are sufficiently similar to a given process model fragment. Note that when it comes to querying over potential executions of process models, process querying techniques that carry out retrieval decisions based on the structure of process model graphs suffer from the same problems raised above in the context of graph databases and graph query languages.

Process querying techniques devised in [94, 95] propose to retrieve process models based on their behavioral profiles [96, 97], i.e., abstract representations of state spaces and/or possible executions of process models. These techniques trade precision of retrieval decisions for efficiency and, thus, cannot be used to implement, or simulate, the querying experience offered by our scenario-based querying approach that operates over all and exact scenarios encoded in process models.

APQL (A Process-model Query Language) [64] is a query language, designed as part of our earlier work om process querying. APQL relies on the use of a finite set of *behavioral predicates* that are grounded in all possible executions of a process model. The work at hand aims at extending our previous ideas on capturing process information needs in the declarative style with imperative statements. Using our scenario-based querying technique, one can specify a request to retrieve process models that induce languages that contain or exclude any given finite language. APQL cannot offer such control over process querying intents. Furthermore, PQL offers a concrete syntax to

express scenario-based querying intents and has an implementation that justifies the feasibility of the approach. Finally, PQL offers a unique approach to the realization of exploratory querying using the label unification principle that, to the best of our knowledge, is unmatched by any other existing process querying technique.

Finally, there exist approaches, like those proposed in [98, 99], tailored for querying finite collections of already observed or currently executing process scenarios. As our querying technique addresses querying of infinite collections of process scenarios, it can be easily adapted for use cases that address querying of finite collections of process executions, e.g., querying of event logs that are actively studied in process mining [56].

10. Conclusion

This paper proposes a method that given a collection of process models and a process scenario template, formally captured using the introduced notion of a trace with wildcards, retrieves models (and their attributes) that describe scenarios that match the template. The method aims to support process compliance, reuse, and standardization use cases by fulfilling their identified requirements of exact scenario matching, partial scenario matching, and activity label similarity. Our evaluation on industrial and synthetic datasets confirms the feasibility of using our approach in industrial settings.

In [33], we proposed a framework for developing process querying methods. The framework is an abstract system in which components can be selectively replaced to result in a new process querying method. According to this framework, the proposed process querying method addresses querying of formal process models specified as net systems, has formal querying semantics, implements the read querying intent (i.e., is designed to retrieve models), has a filtering component that exploits the result captured in Proposition 6.13, and suggests an approach for inspecting and visualizing query results, refer to Section 6.3. According to two recent surveys on process querying methods [34, 33], the proposed method addresses a major gap in process querying.

This paper opens avenues for future work to strengthen and extend the applicability of the proposed method and address its acknowledged limitations.

First, empirical studies need to be conducted to better understand the various requirements of process querying. These studies can help to answer these questions: What should be the expressiveness of queries? Note that the expressiveness of scenario-based querying is limited by the expressiveness of traces with wildcards. How should the query results be presented to the users? Which use cases, beyond process compliance, reuse, and standardization can be supported by scenario-based querying.

Second, we are interested in conducting case studies that aim at assessing the suitability of using scenario-based process querying technique as it is perceived by stakeholders (e.g., business analysts, process experts, and domain experts), for implementing and solving various business problems.

Third, scenario-based process querying can be extended to cater for the infinite trace semantics of process models. This will extend the applicability of the method to process models without a pre-defined terminal state and/or models with behavioral anomalies, e.g., models that only describe non-terminating executions. Another direction for extending the applicability of the technique concerns with designing algorithms for scenario-based querying over general net systems.

Fourth, the proposed technique to explaining results of process querying, refer to Section 6.3, has no control over the number of zero cost moves in alignments that justify executions of traces with wildcards. Hence, the user has no control over which alignments are proposed for explaining the results of querying. Future work can look into providing such control mechanisms. For example, the user may be interested in the shortest possible alignment. An initial idea to provide such control revolves around constructing an optimal alignment as per a cost function that assigns small costs, instead of zero costs used in this work, to moves that are of no interest to the result of the query. This way, optimal alignments will strive to minimize the overall number of moves.

Finally, one may develop further methods that aim at improving the efficiency of the proposed approach. Similar to the index proposed in [100], one can design dedicated data structures that can be pre-computed and later reused to speed up query decisions at run-time. In addition, as suggested in [33], one can think of reusing prior query decisions by introducing caching mechanisms. Note that the discussions of the computability and complexity of our approach in Section 8 are carried for the general problem statement over net systems. However, as future work, one can study the complexity of the scenario-based querying for subclasses of net systems, e.g., workflow systems, and introduce subclass specific optimizations for computing the queries. Also, one should aim at evaluating the performance of the presented querying technique on input net systems, rather than their restricted class of workflow systems, to study the practical limitations of deciding the general form of the trace executability problem, and whether significant differences with the performance over workflow systems are observed.

References

[1] M. Weske, Business Process Management - Concepts, Languages, Architectures, 2nd Edition, Springer, 2012.

[2] M. Dumas, M. L. Rosa, J. Mendling, H. A. Reijers, Fundamentals of Business Process Management, Springer, 2013.

[3] T. Curran, G. Keller, A. Ladd, SAP R/3 Business Blueprint: Understanding the Business Process Reference Model, Prentice-Hall, Inc., 1998.

[4] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, K. Wolf, Analysis on demand: Instantaneous soundness checking of industrial business process models, Data Knowl. Eng. 70 (5) (2011) 448–466.

[5] A. Polyvyanyy, S. Smirnov, M. Weske, Reducing complexity of large EPCs, in: MobIS, Vol. 141 of LNI, GI, 2008, pp. 195–207.

[6] X. Gao, Towards the next generation intelligent BPM - in the era of big data, in: BPM, Vol. 8094 of LNCS, Springer, 2013, pp. 4–9.

[7] W. Damm, D. Harel, LSCs: Breathing life into message sequence charts, Form. Methods Syst. Des. 19 (1) (2001) 45–80.

[8] J. Desel, G. Juhás, R. Lorenz, C. Neumair, Modelling and validation with VipTool, in: BPM, Vol. 2678 of LNCS, Springer, 2003, pp. 380–389.

[9] D. Amyot, A. Eberlein, An evaluation of scenario notations and construction approaches for telecommunication systems development, Telecommun. Syst. 24 (1) (2003) 61–94.

[10] H. Liang, J. Dingel, Z. Diskin, A comparative survey of scenario-based to state-based model synthesis approaches, in: SCESM, ACM, 2006, pp. 5–12.

[11] R. Bergenthum, J. Desel, R. Lorenz, S. Mauser, Synthesis of petri nets from scenarios with VipTool, in: ICATPN, Vol. 5062 of LNCS, Springer, 2008, pp. 388–398.

[12] D. Fahland, Oclets—scenario-based modeling with Petri nets, in: ICATPN, Vol. 5606 of LNCS, Springer, 2009, pp. 223–242.

[13] D. Fahland, From scenarios to components, Ph.D. thesis, Technische Universiteit Eindhoven (2010).

[14] J. Recker, N. Safrudin, M. Rosemann, How novices model business processes, in: BPM, Vol. 6336 of LNCS, Springer, 2010, pp. 29–44.

[15] D. Weitlaner, A. Guettinger, M. Kohlbacher, Intuitive comprehensibility of process models, in: S-BPM ONE, Vol. 360 of CCIS, Springer, 2013, pp. 52–71.

[16] M. Weidlich, A. Polyvyanyy, N. Desai, J. Mendling, M. Weske, Process compliance analysis based on behavioural profiles, Information Systems 36 (7) (2011) 1009–1025.

[17] O. Turetken, A. Elgammal, W.-J. van den Heuvel, M. P. Papazoglou, Capturing compliance requirements: A pattern-based approach, IEEE software 29 (3) (2012) 28–36.

[18] T. Neumuth, F. Loebe, P. Jannin, Similarity metrics for surgical process models, Artificial intelligence in medicine 54 (1) (2012) 15–27.

[19] A. Kumar, W. Yao, C.-H. Chu, Flexible process compliance with semantic constraints using mixed-integer programming, INFORMS Journal on Computing 25 (3) (2013) 543–559.

[20] A. Barnawi, A. Awad, A. Elgammal, R. El Shawi, A. Almalaise, S. Sakr, Runtime self-monitoring approach of business process compliance in cloud environments, Cluster Computing 18 (4) (2015) 1503–1526.

[21] L. T. Ly, F. M. Maggi, M. Montali, S. Rinderle-Ma, W. M. van der Aalst, Compliance monitoring in business processes: Functionalities, application, and tool-support, Information systems 54 (2015) 209–234.

[22] P. Delfmann, M. Steinhorst, H.-A. Dietrich, J. Becker, The generic model query language gmql–conceptual specification, implementation, and runtime evaluation, Information Systems 47 (2015) 129–177.

[23] J. Becker, P. Delfmann, H.-A. Dietrich, M. Steinhorst, M. Eggert, Business process compliance checking–applying and evaluating a generic pattern matching approach for conceptual models in the financial sector, Information Systems Frontiers 18 (2) (2016) 359–405.

[24] A. Elgammal, O. Turetken, W.-J. van den Heuvel, M. Papazoglou, Formalizing and appling compliance patterns for business process compliance, Software & Systems Modeling 15 (1) (2016) 119–146.

[25] D. Knuplesch, M. Reichert, A visual language for modeling multiple perspectives of business process compliance rules, Software & Systems Modeling 16 (3) (2017) 715–736.

[26] P. W. Chung, L. Y. Cheung, C. H. Machin, Compliance flow–managing the compliance of dynamic and complex processes, Knowledge-Based Systems 21 (4) (2008) 332–354.

[27] H. van der Aa, H. Leopold, H. A. Reijers, Checking process compliance against natural language specifications using behavioral spaces, Information Systems 78 (2018) 83–95.

[28] H. Zhuge, A process matching approach for flexible workflow process reuse, Information and Software Technology 44 (8) (2002) 445–450.

[29] R. Xu, P. Lin, Z. Zhao, L. Qian, An approach of reuse-based software process improvement, Journal of Computational Information Systems 6 (6) (2010) 1897–1906.

[30] H. L. Romero, R. M. Dijkman, P. W. P. J. Grefen, A. J. van Weele, A. de Jong, Measures of process harmonization, Inform. Software Tech. 63 (2015) 31–43.

[31] A. Rondini, G. Pezzotta, S. Cavalieri, M.-Z. Ouertani, F. Pirola, Standardizing delivery processes to support service transformation: A case of a multinational manufacturing firm, Computers in Industry 100 (2018) 115–128.

[32] W. M. P. van der Aalst, Business Process Management: A Comprehensive Survey, ISRN Software Engineering 2013.

[33] A. Polyvyanyy, C. Ouyang, A. Barros, W. M. P. van der Aalst, Process querying: Enabling business intelligence through query-based process analytics, Decision Support Systems 100 (2017) 41–56.

[34] J. Wang, T. Jin, R. K. Wong, L. Wen, Querying business process model repositories — A survey of current approaches and issues, World Wide Web 17 (3) (2014) 427–454.

[35] A. Polyvyanyy, L. Corno, R. Conforti, S. Raboczi, M. La Rosa, G. Fortino, Process querying in Apromore, in: BPM Demos, Vol. 1418 of CEUR, CEUR-WS.org, 2015, pp. 105–109.

[36] A. Polyvyanyy, A. H. M. ter Hofstede, M. L. Rosa, C. Ouyang, A. Pika, Process query language: Design, implementation, and evaluation, CoRR abs/1909.09543. arXiv:1909.09543.

[37] C. Baier, J. Katoen, Principles of Model Checking, MIT Press, 2008.

[38] J. Esparza, On the decidability of model checking for several μ-calculi and Petri nets, in: CAAP, Vol. 787 of LNCS, Springer, 1994, pp. 115–129.

[39] A. Ghose, G. Koliadis, Auditing business process compliance, in: ICSOC, Vol. 4749 of LNCS, Springer, 2007, pp. 169–180.

[40] M. Reichert, B. Weber, Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies, Springer, 2012.

[41] S. C. Tosatto, G. Governatori, P. Kelsen, Business process regulatory compliance is hard, IEEE T. Serv. Comput. 8 (6) (2015) 958–970.

[42] A. Koschmider, M. Fellmann, A. Schoknecht, A. Oberweis, Analysis of process model reuse: Where are we now, where should we go from here?, Decis. Support Syst. 66 (2014) 9–19.

[43] L. Aldin, S. de Cesare, A literature review on business process modelling: New frontiers of reusability, Enterp. Inf. Syst. 5 (3) (2011) 359–383.

[44] W. B. Frakes, K. Kang, Software reuse research: Status and future, IEEE Trans. Software Eng. 31 (7) (2005) 529–536.

[45] D. Grigori, J. C. Corrales, M. Bouzeghoub, A. Gater, Ranking BPEL processes for service discovery, IEEE T. Serv. Comput. 3 (3) (2010) 178–192.

[46] M. Lincoln, M. Golani, A. Gal, Machine-assisted design of business process models using descriptor space analysis, in: BPM, Vol. 6336 of LNCS, Springer, 2010, pp. 128–144.

[47] A. Awad, S. Sakr, M. Kunze, M. Weske, Design by selection: A reuse-based approach for business process modeling, in: ER, Vol. 6998 of LNCS, Springer, 2011, pp. 332–345.

[48] N. C. Narendra, K. Ponnalagu, G. Gangadharan, H. L. Truong, S. Dustdar, A. K. Ghose, Effective reuse via modeling, managing and searching of business process assets, in: IEEE SCC, IEEE, 2012, pp. 462–469.

[49] M. Fantinato, M. B. F. de Toledo, L. H. Thom, I. M. de Souza Gimenes, R. dos Santos Rocha, D. Z. G. Garcia, A survey on reuse in the business process management domain, Int. J. Business Process Integration and Management 6 (1) (2012) 52–76.

[50] R. Tregear, Handbook on Business Process Management: Part II, Springer Berlin Heidelberg, 2010, Ch. Business Process Standardization, pp. 307–327.

[51] M. L. Rosa, M. Dumas, C. C. Ekanayake, L. García-Bañuelos, J. Recker, A. H. M. ter Hofstede, Detecting approximate clones in business process model repositories, Inf. Syst. 49 (2015) 102–125.

[52] J. vom Brocke, A. Simons, B. Niehaves, K. Riemer, R. Plattfaut, A. Cleven, Reconstructing the giant: On the importance of rigour in documenting the literature search process, in: ECIS, 2009, pp. 2206–2217.
URL http://aisel.aisnet.org/ecis2009/161

[53] L. T. Ly, S. Rinderle-Ma, K. Göser, P. Dadam, On enabling integrated process compliance with semantic constraints in process management systems, Information Systems Frontiers 14 (2) (2012) 195–219.

[54] W. Reisig, Elements of Distributed Algorithms: Modeling and Analysis with Petri nets, Springer, 1998.

[55] W. M. P. van der Aalst, Verification of workflow nets, in: ICATPN, Vol. 1248 of LNCS, Springer, 1997, pp. 407–426.

[56] W. M. P. van der Aalst, Process Mining—Data Science in Action, Second Edition, Springer Berlin Heidelberg, 2016.

[57] W. M. P. van der Aalst, A. Adriansyah, B. F. van Dongen, Replaying history on process models for conformance checking and performance analysis, WIREs Data Mining and Knowledge Discovery 2 (2) (2012) 182–192.

[58] A. Adriansyah, Aligning observed and modeled behavior, Ph.D. thesis, TU/e (2014).

[59] R. van der Toorn, Component-based Software Design with Petri Nets: an Approach Based on Inheritance of Behavior, Beta Dissertation, Technische Universiteit Eindhoven, 2004.

[60] A. Awad, A. Polyvyanyy, M. Weske, Semantic querying of business process models, in: ECOC, IEEE Computer Society, 2008, pp. 85–94.

[61] B. Meyer, Introduction to the Theory of Programming Languages, Prentice-Hall, 1990.

[62] C. Date, H. Darwen, A Guide to the SQL Standard: A User's Guide to the Standard Database Language SQL, Addison-Wesley, 1997.

[63] A. Polyvyanyy, Structuring process models, Ph.D. thesis, University of Potsdam (2012).

[64] A. H. M. ter Hofstede, C. Ouyang, M. La Rosa, L. Song, J. Wang, A. Polyvyanyy, APQL: A process-model query language, in: AP-BPM, Vol. 159 of LNBIP, Springer, 2013, pp. 23–38.

[65] Object Management Group (OMG), Business Process Model and Notation (BPMN), Version 2.0, OMG Document Number formal/2011-01-03 (`http://www.omg.org/spec/BPMN/2.0/`) (2011).

[66] R. M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inform. Software Tech. 50 (12) (2008) 1281–1294.

[67] A. Polyvyanyy, M. Weidlich, R. Conforti, M. La Rosa, A. H. M. ter Hofstede, The 4C spectrum of fundamental behavioral relations for concurrent systems, in: Petri Nets, Vol. 8489 of LNCS, Springer, 2014, pp. 210–232.

[68] T. J. Parr, The Definitive ANTLR 4 Reference, Oreilly and Associate Series, Pragmatic Programmers, LLC, 2013.

[69] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, W. M. P. van der Aalst, The ProM framework: A new era in process mining tool support, in: ICATPN, Vol. 3536 of LNCS, Springer, 2005, pp. 444–454.

[70] A. Polyvyanyy, Evaluation of a novel information retrieval model: eTVSM, Master's thesis, University of Potsdam (2007).

[71] Y. Li, B. Liu, A normalized Levenshtein distance metric, EEE Trans. Pattern Anal. Mach. Intell. 29 (6) (2007) 1091–1095.

[72] A. Polyvyanyy, L. García-Bañuelos, D. Fahland, M. Weske, Maximal structuring of acyclic process models, The Computer Journal 57 (1) (2014) 12–35.

[73] B. Kiepuszewski, A. H. M. ter Hofstede, W. M. P. van der Aalst, Fundamentals of control flow in workflows, Acta Inf. 39 (3) (2003) 143–209.

[74] Z. Yan, R. M. Dijkman, P. W. Grefen, Generating process model collections, Softw. Syst. Model. (2015) 1–17.

[75] K. Schmidt, LoLA: A low level analyser, in: ICATPN, Vol. 1825 of LNCS, Springer, 2000, pp. 465–474.

[76] M. Felleisen, On the expressive power of programming languages, Sci. Comput. Program. 17 (1–3) (1991) 35–75.

[77] A. Polyvyanyy, A. Armas-Cervantes, M. Dumas, L. García-Bañuelos, On the expressive power of behavioral profiles, Formal Asp. Comput. 28 (4) (2016) 597–613.

[78] K. Wolf, Interleaving based model checking of concurrency and causality, Fundamenta Informaticae 161 (4) (2018) 423–445.

[79] C. Reutenauer, The Mathematics of Petri Nets, Prentice-Hall, Inc., 1990.

[80] R. Lipton, The Reachability Problem Requires Exponential Space, Research report, Department of Computer Science, Yale University, 1976.

[81] A. Polyvyanyy, Encyclopedia of Big Data Technologies, Springer International Publishing, 2018, Ch. Business Process Querying, pp. 1–9.

[82] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J. D. Reese, Model checking large software specifications, IEEE Trans. Software Eng. 24 (7) (1998) 498–520.

[83] A. Gurfinkel, M. Chechik, B. Devereux, Temporal logic query checking: A tool for model exploration, IEEE Trans. Software Eng. 29 (10) (2003) 898–914.

[84] R. M. Karp, R. E. Miller, Parallel program schemata, J. Comput. Syst. Sci. 3 (2) (1969) 147–195.

[85] R. Angles, C. Gutiérrez, Survey of graph database models, ACM Comput. Surv. 40 (1).

[86] P. T. Wood, Query languages for graph databases, ACM SIGMOD Record 41 (1) (2012) 50–60.

[87] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty years of graph matching in pattern recognition, Int. J. Pattern Recogn. 18 (3) (2004) 265–298.

[88] P. Barceló, L. Libkin, A. W. Lin, P. T. Wood, Expressive languages for path queries over graph-structured data, ACM T. Database Syst. 37 (4).

[89] X. Zhao, C. Xiao, X. Lin, W. Wang, Y. Ishikawa, Efficient processing of graph similarity queries with edit distance constraints, The VLDB Journal 22 (6) (2013) 727–752.

[90] A. Valmari, The state explosion problem, in: Lectures on Petri Nets I: Basic Models, Vol. 1491 of LNCS, Springer, 1998, pp. 429–528.

[91] C. Beeri, A. Eyal, S. Kamenkovich, T. Milo, Querying business processes with BP-QL, Inf. Syst. 33 (6) (2008) 477–507.

[92] A. Awad, S. Sakr, On efficient processing of BPMN-Q queries, Comput. Ind. 63 (9) (2012) 867–881.

[93] R. M. Dijkman, M. Dumas, B. F. van Dongen, R. Käärik, J. Mendling, Similarity of business process models: Metrics and evaluation, Inf. Syst. 36 (2) (2011) 498–516.

[94] T. Jin, J. Wang, L. Wen, Querying business process models based on semantics, in: DASFAA, Vol. 6588 of LNCS, Springer, 2011, pp. 164–178.

[95] M. Kunze, M. Weidlich, M. Weske, Querying process models by behavior inclusion, Softw. Syst. Model. 14 (3) (2015) 1105–1125.

[96] M. Weidlich, A. Polyvyanyy, J. Mendling, M. Weske, Causal behavioural profiles – efficient computation, applications, and evaluation, Fund. Inform. 113 (3-4) (2011) 399–435.

[97] M. Weidlich, J. Mendling, M. Weske, Efficient consistency measurement based on behavioral profiles of process models, IEEE Trans. Software Eng. 37 (3) (2011) 410–429.

[98] D. Deutch, T. Milo, Type inference and type checking for queries over execution traces, The VLDB Journal 21 (1) (2012) 51–68.

[99] C. Beeri, A. Eyal, T. Milo, A. Pilberg, Monitoring business processes with queries, in: ICVLDB, ACM, 2007, pp. 603–614.

[100] A. Polyvyanyy, M. La Rosa, A. H. M. ter Hofstede, Indexing and efficient instance-based retrieval of process models using untanglings, in: CAiSE, Vol. 8484 of LNCS, Springer, 2014, pp. 439–456.

Appendix A. Proofs

This appendix contains proofs of formal statements claimed in the paper. Let σ be a sequence. By $\mathit{prefix}(\sigma, i)$, $i \in [0..|\sigma|] \cap \mathbb{N}_0$, we denote the prefix of σ up to and including position i. For example, if $\sigma = \langle \mathtt{a},\mathtt{b},\mathtt{a},\mathtt{b},\mathtt{a},\mathtt{h},\mathtt{a},\mathtt{l},\mathtt{a},\mathtt{m},\mathtt{a},\mathtt{h},\mathtt{a}\rangle$, then $\mathit{prefix}(\sigma,5) = \langle \mathtt{a},\mathtt{b},\mathtt{a},\mathtt{b},\mathtt{a}\rangle$. We say that a sequence η is a prefix of a sequence ρ iff there exists $n \in \mathbb{N}_0$ such that $\eta = \mathit{prefix}(\rho, n)$.

Lemma 6.5 (Trace equivalence for labels unification)**.**
Let $(\hat{S}, g)$, $\hat{S} := (\hat{N}, M_{ini}, M_{fin})$, $\hat{N} := (\hat{P}, \hat{T}, \hat{F}, \hat{\Lambda}, \hat{\lambda})$, be a result of sets of labels unification in a net system $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, for $\Delta \subseteq \wp(\mathbb{\Lambda})$, and let $\eta \in \hat{\Lambda}^*$. There exists $\rho \in L(S)$ such that $\eta \in \mathit{expand}(\rho, g)$ iff $\eta \in L(\hat{S})$. ⌟

Proof. Let $\langle (S_1, \alpha_1), \ldots, (S_n, \alpha_n)\rangle$, $n = |\Delta|$, be a sequence of labels unifications used to implement the sets of labels unification, i.e., $\hat{S} = S_n$, cf. Definition 6.3, where (S_i, α_i), $i \in [1..n]$, is a labels unification for $\Delta_i \in \Delta$. We prove each part of the statement separately.

($\Rightarrow$) Next, we demonstrate that if there exists $\rho \in L(S)$ such that $\eta \in \mathit{expand}(\rho, g)$, then $\eta \in L(\hat{S})$. Let $\rho \in L(S)$ and let $\eta \in \mathit{expand}(\rho, g)$. Let $\sigma \in \mathbb{E}_S$ be an execution of S that induces label sequence ρ. Let $\hat{\sigma} := \mathit{construct}(\sigma, f, \eta)$ be a sequence constructed from σ, f, and η, where $f := \{(x, \langle \acute{x}^\alpha, \hat{\imath}^\alpha, \grave{x}^\alpha\rangle) \in T \times \hat{T}^* \mid \exists\, i \in [1..|\Delta|] : \lambda(x) \in \Delta_i \wedge \alpha = \alpha_i\}$ and the *construct* function, $\mathit{construct} : T^* \times (T \times \hat{T}^*) \times \hat{\Lambda}^* \to \hat{T}^*$, is defined as suggested below[14]:

[14] For the example sets of labels unification in the net system in Figure 7 shown in Figure 8, it holds that $f = \{(t_2, \langle \acute{t}_2^\kappa, \hat{\imath}^\kappa, \grave{\imath}_2^\kappa\rangle), (t_2, \langle \acute{t}_2^\mu, \hat{\imath}^\mu, \grave{\imath}_2^\mu\rangle), (t_3, \langle \acute{t}_3^\kappa, \hat{\imath}^\kappa, \grave{\imath}_3^\kappa\rangle), (t_4, \langle \acute{t}_4^\mu, \hat{\imath}^\mu, \grave{\imath}_4^\mu\rangle)\}$.

$$construct(x,y,z) := \begin{cases} \langle\rangle & x \text{ is empty} \\ u \circ h(u,x,y,z), \text{where} & \text{otherwise.} \\ u = construct(v,y,z), & \\ v = prefix(x,|x|-1) & \end{cases}$$

$$h(\chi,x,y,z) := \begin{cases} \langle x_{[|x|]} \rangle & \hat{\lambda}(\chi \circ \langle x_{[|x|]} \rangle)|_{\hat{\Lambda}} \\ & \text{is a prefix of } z \\ \langle \acute{w}^{\alpha}, \hat{t}^{\alpha}, \grave{w}^{\alpha} \rangle, \text{where} & \text{otherwise.} \\ (w, \langle \acute{w}^{\alpha}, \hat{t}^{\alpha}, \grave{w}^{\alpha} \rangle) \in y, w = x_{[|x|]} & \\ \alpha = z_{[i]}, i = |(\hat{\lambda}(\chi)|_{\hat{\Lambda}})| + 1 & \end{cases}$$

Then, by structural induction on σ, it holds that $\hat{\sigma}$ is an occurrence sequence of $\hat{S}$ that induces a label sequence which is a prefix of η. In the base case, it holds that $construct(\langle\rangle, f, \eta) = \langle\rangle$ and, clearly, $\langle\rangle$ is a prefix of η. Assume that for some $n \in [0..|\sigma|)$ it holds that $construct(prefix(\sigma,n), f, \eta)$ induces a label sequence which is a prefix of η. Then, it also holds that $construct(prefix(\sigma,n+1), f, \eta)$ induces a prefix of η. Note that $construct(prefix(\sigma,n+1), f, \eta) := construct(prefix(\sigma,n), f, \eta) \circ h(construct(prefix(\sigma,n), f, \eta), prefix(\sigma,n+1), f, \eta)$ and the result of function h is either a sequence of one silent transition or a sequence of transitions r such that $\hat{\lambda}(r)|_{\hat{\Lambda}} = \langle \eta_{[j]} \rangle$, where $j = |(\hat{\lambda}(construct(prefix(\sigma,n), f, \eta))|_{\hat{\Lambda}})| + 1$. It is easy to see that both σ and $\hat{\sigma}$ have the same number of observable transitions and, hence, the label sequence induced by $\hat{\sigma}$ is equal to η; note that $|\rho| = |\eta|$. Finally, by construction of $\hat{\sigma}$ and net system $\hat{S}$, refer to Definitions 6.1 and 6.3, it holds that $\hat{\sigma} \in \mathbb{E}_{\hat{S}}$; by induction, for every $i \in [1..|\sigma|)$ it holds that $prefix(\sigma,i)$ and $construct(\sigma, f, \eta)$ lead to the same marking.

($\Leftarrow$) Next, we demonstrate that if $\eta \in L(\hat{S})$, then there exists $\rho \in L(S)$ such that $\eta \in expand(\rho, g)$. Let $\hat{\sigma} \in \mathbb{E}_{\hat{S}}$ such that $\hat{\lambda}(\hat{\sigma})|_{\hat{\Lambda}}$. Let $\sigma := rewrite(\hat{\sigma}|_{T \cup X}, k)$, where X is the set of all the presolitary transitions of $\hat{N}$ introduced by labels unifications that resulted in (S_i, α_i), $i \in [1..n]$, and $k := \{(x, \langle y \rangle) \in X \times T^* \mid x \text{ is the presolitary transition of } y\}$. By construction, $\sigma \in \mathbb{E}_S$, $\lambda(\sigma)|_{\Lambda} \in L(S)$, and $\eta \in expand(\lambda(\sigma)|_{\Lambda}, g)$.

∎

Lemma 6.10 (Trace equivalence for sequences test)**.**
Let $(\hat{S}, g)$, $\hat{S} := (\hat{N}, \hat{M}_{ini}, \hat{M}_{fin})$, $\hat{N} := (\hat{P}, \hat{T}, \hat{F}, \hat{\Lambda}, \hat{\lambda})$, *be a result of sequences test insertion in a net system* $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, *for* $\Phi \subseteq \Lambda^*$. *Let* $\rho \in \Lambda^*$ *and* $\eta \in \hat{\Lambda}^*$ *such that* $\rho = rewrite(\eta, g)$. *Then,* $\rho \in L(S)$ *iff* $\eta \in L(\hat{S})$.

Proof. We prove each part of the statement separately.

($\Rightarrow$) Next, we demonstrate that if $\rho \in L(S)$, then it holds that $\eta \in L(\hat{S})$. Let $\sigma \in \mathbb{E}_S$ be an execution of S that induces label sequence ρ, i.e., it holds that $\lambda(\sigma)|_{\Lambda} = \rho$. Note that by construction of $\hat{S}$, it holds that σ is also an execution of $\hat{S}$, i.e., $\sigma \in \mathbb{E}_{\hat{S}}$. Moreover, by induction on the size of the prefix of σ, for every $i \in [0..|\sigma|]$ it holds that $(\hat{N}, M_{ini} \uplus$

$[\hat{p}])[prefix(\sigma,i)\rangle(\hat{N}, M_i \uplus [\hat{p}])$, where M_i is such that $(N, M_{ini})[prefix(\sigma,i)\rangle(N, M_i)$. Hence, because $prefix(\sigma,|\sigma|) = \sigma$, it holds that $(\hat{N}, \hat{M}_{ini})[\sigma\rangle(\hat{N}, \hat{M}_{fin})$; note that $\hat{M}_{fin} = M_{fin} \uplus [\hat{p}]$ and $M_{|\sigma|} = M_{fin}$. Let us assume that $\eta \notin L(\hat{S})$. Then, for every execution x of $\hat{S}$, it holds that $\hat{\lambda}(x)|_{\hat{\Lambda}} \neq \eta$. Let function $construct : \mathbb{A}^* \times \mathbb{N}_0 \times T^* \to \hat{T}^*$ be defined as suggested below:

$$construct(x,y,z) := \begin{cases} \langle\rangle & y = 0 \\ construct(x,y-1,z) \circ tail(x,y,z) & y > 0 \wedge x_{[y]} \notin dom(g) \\ construct(x,y-1,z) \circ \\ \quad map(tail(x,y,z), x_{[y]}) & y > 0 \wedge x_{[y]} \in dom(g) \end{cases}$$

where function $tail : \mathbb{A}^* \times \mathbb{N}_0 \times T^* \to \hat{T}^*$ is defined as follows $tail(x,y,z) := suffix(sprefix(z, rewrite(prefix(x,y),g)), |construct(x,y-1,z)|+1)$, $sprefix : T^* \times \mathbb{A}^* \to T^*$ is such that $sprefix(u,v)$ is the smallest prefix of u that induces v, and $map : T^* \times \mathbb{A} \to \hat{T}^*$ is such that $map(u,w) := rewrite(u, f(w))$, where $f(w) := \{(a, \langle b\rangle) \in T \times \hat{T}^* \mid \exists\, i \in [1\,..\,|g(w)|] : (a = tr(N, g(w)_{[i]}) \wedge b = t_{(g(w),i)})\}$; note that every label in every sequence in Φ is a sole label in N and every transition $t_{(\phi,i)}$, where $\phi \in \Phi$ and $i \in [1\,..\,|\phi|]$, is a fresh transition of $\hat{N}$ that mimics transition $tr(N, \phi_{[i]})$. For every $i \in [0\,..\,|\eta|]$ it holds that $\hat{\lambda}(construct(\eta,i,\sigma))|_{\hat{\Lambda}}$ is a prefix of η. Therefore, it holds that $\hat{\lambda}(construct(\eta,|\eta|,\sigma))|_{\hat{\Lambda}} = \eta$; it is easy to see that $|(\hat{\lambda}(construct(\eta,i,\sigma))|_{\hat{\Lambda}})| = i$. By induction on the length of the prefix of η, for each $i \in [0\,..\,|\eta|]$ it holds that $(\hat{N}, \hat{M}_{ini})[prefix(\sigma, |construct(\eta,i,\sigma)|)\rangle(\hat{N}, M)$ and $(\hat{N}, \hat{M}_{ini})[construct(\eta,i,\sigma)\rangle(\hat{N}, M)$, i.e., both $construct(\eta,i,\sigma)$ and the prefix of σ of the size of $construct(\eta,i,\sigma)$ lead to the same marking. Moreover, it holds that $\hat{\lambda}(suffix(\sigma, |(construct(\eta,|\eta|, \sigma))|+1))|_{\hat{\Lambda}} = \varepsilon$. Thus, $\hat{\sigma} \in \mathbb{E}_{\hat{S}}$ and $\hat{\lambda}(\hat{\sigma})|_{\hat{\Lambda}} = \eta$, where $\hat{\sigma} := construct(\eta,|\eta|,\sigma) \circ suffix(\sigma, |(construct(\eta,|\eta|,\sigma))|+1)$.

($\Leftarrow$) Next, we demonstrate that if $\eta \in L(\hat{S})$, then $\rho \in L(S)$. Let $\hat{\sigma} \in \mathbb{E}_{\hat{S}}$ be an execution of $\hat{S}$ that induces label sequence η, i.e., it holds that $\hat{\lambda}(\hat{\sigma})|_{\hat{\Lambda}} = \eta$. Let $\sigma := rewrite(\hat{\sigma}, f)$, where $f := \{(x, \langle y\rangle) \in \hat{T} \times \hat{T}^* \mid \exists\, \phi \in \Phi\, \exists\, i \in [1\,..\,|\phi|] : x = t_{(\phi,i)} \wedge y = tr(\hat{N}, \phi_{[i]})\}$; note that every label in every sequence in Φ is a sole label in $\hat{N}$ and every transition $t_{(\phi,i)}$, where $\phi \in \Phi$ and $i \in [1\,..\,|\phi|]$, is a fresh transition of $\hat{N}$ that mimics transition $tr(\hat{N}, \phi_{[i]})$ of N.[15] It holds that σ is an occurrence sequence of $\hat{S}$. First, by construction of σ, it holds that $|\sigma| = |\hat{\sigma}|$. Second, by structural induction on prefixes of σ, for every $i \in [0\,..\,|\sigma|]$ it holds that $(\hat{M}_i \setminus X) \uplus [\hat{p}] = M_i$, where M_i, X, and $\hat{M}_i$ are such that $(\hat{N}, \hat{M}_{ini})[prefix(\sigma,i)\rangle(\hat{N}, M_i)$, $(\hat{N}, \hat{M}_{ini})[prefix(\hat{\sigma},i)\rangle(\hat{N}, \hat{M}_i)$, and $X = \hat{P} \setminus P$. If $i = |\sigma|$, then $((M_{fin} \uplus [\hat{p}]) \setminus X) \uplus [\hat{p}] = M_{|\sigma|} = M_{fin} \uplus [\hat{p}]$; note that $\hat{M}_{|\sigma|} = M_{fin} \uplus [\hat{p}]$. Therefore, σ leads to $\hat{M}_{fin} = M_{fin} \uplus [\hat{p}]$, i.e., it holds that $(\hat{N}, \hat{M}_{ini})[\sigma\rangle(\hat{N}, \hat{M}_{fin})$ and, thus, σ is an execution of $\hat{S}$. Note that it also holds that σ is an execution of S. By structural induction on prefixes of σ, for every $j \in [0\,..\,|\sigma|]$ it also holds that $\hat{M}_j \setminus [\hat{p}] = M_j$, where M_j and $\hat{M}_j$ are such that $(\hat{N}, \hat{M}_{ini})[prefix(\sigma,j)\rangle(\hat{N}, \hat{M}_j)$ and

[15] For the example sequences test insertion shown in Figure 11, it holds that $f = \{(t_{(u,1)}, \langle t_4\rangle), (t_{(u,2)}, \langle t_5\rangle), (t_{(u,3)}, \langle t_4\rangle), (t_{(v,1)}, \langle t_5\rangle), (t_{(v,2)}, \langle t_7\rangle)\}$, where $u = \texttt{cbc}$ and $v = \texttt{bd}$.

$(N, M_{ini})[prefix(\sigma, j)\rangle(N, M_j)$. Hence, $(N, M_{ini})[\sigma\rangle(N, M_{fin})$ because $\hat{M}_{|\sigma|} = \hat{M}_{fin}$ and $\hat{M}_{fin} \setminus [\hat{p}] = M_{fin}$. Finally, it holds that $\lambda(\sigma)|_\Lambda = \rho$. Let $u := \langle 0 \rangle \circ v$, where v is a sequence of positions in $\hat{\sigma}$ (in ascending order) that hold observable transitions. Then, for every $k \in [1..|u|)$ it holds that $prefix(\sigma, u_{[k]})|_\Lambda \circ tail(u_{[k+1]}) = rewrite(prefix(\eta, k), g) = prefix(\sigma, u_{[k+1]})|_\Lambda$, such that $tail(x) := \left\langle \hat{\lambda}(\hat{\sigma}_{[x]}) \right\rangle$ if $\hat{\lambda}(\hat{\sigma}_{[x]}) \notin dom(g)$, and $tail(x) := g(\hat{\lambda}(\hat{\sigma}_{[x]}))$ otherwise. Finally, by construction of σ, for every position $i \in (u_{|u|}..|\sigma|]$ in σ it holds that $\lambda(\sigma_{[i]})$ is a silent transition.

■

Theorem 6.11 (Trace executability)**.**
A workflow system $S := (N, M_{ini}, M_{fin})$, $N := (P, T, F, \Lambda, \lambda)$, executes a trace with wildcards ω iff there exists $\eta \in L(\hat{S})$, $rewrite(\eta|_{dom(h)}, h) = (\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)|_{\mathbb{A}}$, where:

- *$(\hat{S}, h)$ is a result of sequences test insertion in S'' for $maxsubseq(\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)$,*
- *(S'', α, ζ) is a result of framing S',*
- *(S', g) is a result of sets of labels unification in S for $\{X \subseteq \mathbb{A} \mid \exists i \in [1..|\omega|] : X = M_{\texttt{Event}}(\omega_{[i]})\}$, and*
- *$f(e) := \left\langle g^{-1}(M_{\texttt{Event}}(e)) \right\rangle$, $e \in Set(\omega) \setminus \{*\}$.*

Proof. We prove each part of the statement separately.

($\Rightarrow$) Next, we demonstrate that if S executes ω, then there exists $\eta \in L(\hat{S})$ such that $rewrite(\eta|_{dom(h)}, h) = (\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)|_{\mathbb{A}}$. If S executes ω, then $L(S) \cap L(\omega) \neq \emptyset$, refer to Definition 4.4. Let $\rho \in L(S) \cap L(\omega)$ be a string. We construct $\eta \in \hat{\Lambda}^*$ from ρ in three steps. First, let $x \in expand(\rho, g)$ be such that x contains all the maximal label substrings of $rewrite(\omega, f)$ in the order they appear in $rewrite(\omega, f)$. Note that if the first (the last) element of $rewrite(\omega, f)$ is not the special $*$ symbol, then it must be matched with the first (the last) symbol of x. Because $\rho \in L(\omega)$, x always exists. Second, let $y \in \hat{\Lambda}^*$ be such that $y = \langle \alpha \rangle \circ x \circ \langle \zeta \rangle$. Third, let $\eta \in \hat{\Lambda}^*$ be such that $y = rewrite(\eta, h)$, there exists a bijection k between positions of s and some positions of η, where s is a sequence of maximal label subsequences of $z := \langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle$ in the order they appear in z, for which it holds that for every $u \in dom(k)$, $\eta_{[k(u)]} \in dom(h)$ and $h(\eta_{[k(u)]}) = s_{[u]}$, for every two positions i and j of s, $i < j$, it holds that $k(i) < k(j)$, symbols from $dom(h)$ appear in η only at positions in $img(k)$, $k(1) = 1$, and $k(|s|) = |\eta|$. Because $\rho \in L(\omega)$ and the construction of y, η always exists.[16] According to Lemma 6.5, it holds that $x \in L(S')$. According to Lemma 6.8, it holds that $y \in L(S'')$. According to

[16] Consider the example transformations shown in Figure 12. Let $\rho = \texttt{abcdebcdgfi}$; note that $\rho \in L(S) \cap L(\omega)$. Recall from Section 6.2 that ω is given by $\langle (\texttt{a}, 1.0), *, (\texttt{d}, 1.0), (\texttt{e}, 1.0), *, (\texttt{j}, 0.75), (\texttt{f}, 1.0), * \rangle$ and $g = \{(\texttt{a}, \{\texttt{a}\}), (\texttt{d}, \{\texttt{d}\}), (\texttt{e}, \{\texttt{e}\}), (\mu, \{\texttt{g}, \texttt{h}\}), (\texttt{f}, \{\texttt{f}\})\}$; $M_{\texttt{Event}}(\omega_{[6]}) = \{\texttt{g}, \texttt{h}\}$. Then, $f = \{(\omega_{[1]}, \langle \texttt{a} \rangle), (\omega_{[3]}, \langle \texttt{d} \rangle), (\omega_{[4]}, \langle \texttt{e} \rangle), (\omega_{[6]}, \langle \mu \rangle), (\omega_{[7]}, \langle \texttt{f} \rangle)\}$ and $rewrite(\omega, f) = \langle \texttt{a}, *, \texttt{d}, \texttt{e}, *, \mu, \texttt{f}, * \rangle$. Hence, it holds that $x = \underline{\texttt{a}}\texttt{bc}\underline{\texttt{de}}\texttt{bcd}\underline{\mu}\underline{\texttt{f}}\texttt{i}$. Note that the underlined substrings in x follow the order in which these substrings appear in $rewrite(\omega, f)$. Also, note that $x \in expand(\rho, g)$. Finally, $y = \alpha\texttt{abcdebcd}\mu f i \zeta$ and $\eta = \chi\texttt{bc}\phi\texttt{bcd}\psi\texttt{i}\theta$, where $y = rewrite(\eta, h)$, $h = \{(\chi, \langle \alpha, \texttt{a} \rangle), (\phi, \langle \texttt{d}, \texttt{e} \rangle), (\psi, \langle \mu, \texttt{f} \rangle), (\theta, \langle \zeta \rangle)\}$, $z = \langle \alpha, \texttt{a}, *, \texttt{d}, \texttt{e}, *, \mu, \texttt{f}, *, \zeta \rangle$, $s = \langle \langle \alpha, \texttt{a} \rangle, \langle \texttt{d}, \texttt{e} \rangle, \langle \mu, \texttt{f} \rangle, \langle \zeta \rangle \rangle$, and k is such that $k(1) = 1$, $k(2) = 4$, $k(3) = 8$, and $k(4) = 10$.

Lemma 6.10, it holds that $\eta \in L(\hat{S})$. Finally, by construction of η from ρ, it holds that $rewrite(\eta|_{dom(h)}, h) = (\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle)|_{\not\Lambda}$.

($\Leftarrow$) Next, we demonstrate that if there exists $\eta \in L(\hat{S})$ such that $rewrite(\eta|_{dom(h)}, h) = (\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle)|_{\not\Lambda}$, then S executes ω. Let $\eta \in L(\hat{S})$ be such that $rewrite(\eta|_{dom(h)}, h) = (\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle)|_{\not\Lambda}$. We construct $\rho \in \Lambda^*$ from η in three steps. First, let $x := rewrite(\eta, h)$. Note that x contains all the maximal label substrings of $\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle$ in the order they appear in $\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle$. Moreover, α and ζ are the first and the last symbols of x, respectively. Second, let $y \in \hat{\Lambda}^*$ be such that $x = \langle\alpha\rangle \circ y \circ \langle\zeta\rangle$. Note that y contains all the maximal label substrings of $rewrite(\omega, f)$ in the order they appear in $rewrite(\omega, f)$. Moreover, if the first (the last) element of $rewrite(\omega, f)$ is not the special $*$ symbol, then it is matched with the first (the last) symbol of y. According to Lemma 6.10, it holds that $x \in L(S'')$. According to Lemma 6.8, it holds that $y \in L(S')$. Then, according to Lemma 6.5, there exists $\rho \in L(S)$ such that $y \in expand(\rho, g)$. Thus, in the third step, we select $\rho \in \Lambda^*$ for which it holds that $y \in expand(\rho, g)$.[17] By construction, it also holds that $\rho \in L(\omega)$.

∎

Lemma 6.12 (Trace executability)**.**
There exists $\eta \in L(\hat{S})$ *such that* $rewrite(\eta|_{dom(h)}, h) = (\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle)|_{\not\Lambda}$ *iff* $c(\gamma) = 0$*, where*:

- $(\hat{S}, h)$ *is a result of sequences test insertion in* S'' *for* $maxsubseq(\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle)$*,*
- (S'', α, ζ) *is a result of framing* S'*,*
- (S', g) *is a result of sets of labels unification in* S *for* $\{X \subseteq \not\Lambda \mid \exists i \in [1..|\omega|] : X = M_{\texttt{Event}}(\omega_{[i]})\}$*,*
- $f(e) := \left\langle g^{-1}(M_{\texttt{Event}}(e))\right\rangle$*,* $e \in Set(\omega) \setminus \{*\}$*,*
- S *is a workflow system,*
- ω *is a trace with wildcards,*
- γ *is an optimal alignment between* ρ *and* $\hat{S}$*,*
- ρ *is a finite sequence of symbols over* $dom(h)$ *such that* $rewrite(\rho, h) = (\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle)|_{\not\Lambda}$*, and*
- c *is the move on trace cost function over* $\hat{S}$*.*

⌟

Proof. We prove each part of the statement separately.

($\Rightarrow$) Next, we demonstrate that if there exists $\eta \in L(\hat{S})$ such that $rewrite(\eta|_{dom(h)}, h) = (\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle)|_{\not\Lambda}$, then $c(\gamma) = 0$. Let $\eta \in L(\hat{S})$ be such that $rewrite(\eta|_{dom(h)}, h) = (\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle)|_{\not\Lambda}$. Let $\hat{\sigma} \in \mathbb{E}_{\hat{S}}$ be such that $\hat{\lambda}(\hat{\sigma})|_{\hat{\Lambda}} = \eta$. Then, $c(x) = 0$, where x is an optimal alignment between $\eta|_{dom(h)}$ and $\hat{\sigma}$; note

[17] Consider the example transformations shown in Figure 12. Let $\eta = \chi\texttt{bc}\phi\texttt{bcd}\psi\texttt{i}\theta$; note that $\eta \in L(\hat{S})$ and it holds that $rewrite(\eta|_{dom(h)}, h) = (\langle\alpha\rangle \circ rewrite(\omega, f) \circ \langle\zeta\rangle)|_{\not\Lambda}$. Recall from Section 6.2 that ω is given by $\langle(\texttt{a}, 1.0), *, (\texttt{d}, 1.0), (\texttt{e}, 1.0), *, (\texttt{j}, 0.75), (\texttt{f}, 1.0), *\rangle$ and $g = \{(\texttt{a}, \{\texttt{a}\}), (\texttt{d}, \{\texttt{d}\}), (\texttt{e}, \{\texttt{e}\}), (\mu, \{\texttt{g}, \texttt{h}\}), (\texttt{f}, \{\texttt{f}\})\}$; $M_{\texttt{Event}}(\omega_{[6]}) = \{\texttt{g}, \texttt{h}\}$. Then, $f = \{(\omega_{[1]}, \langle\texttt{a}\rangle), (\omega_{[3]}, \langle\texttt{d}\rangle), (\omega_{[4]}, \langle\texttt{e}\rangle), (\omega_{[6]}, \langle\mu\rangle), (\omega_{[7]}, \langle\texttt{f}\rangle)\}$, $rewrite(\omega, f) = \langle\texttt{a}, *, \texttt{d}, \texttt{e}, *, \mu, \texttt{f}, *\rangle$. Moreover, $h = \{(\chi, \langle\alpha, \texttt{a}\rangle), (\phi, \langle\texttt{d}, \texttt{e}\rangle), (\psi, \langle\mu, \texttt{f}\rangle), (\theta, \langle\zeta\rangle)\}$. Then, $x = \alpha\texttt{abcdebcd}\mu\texttt{fi}\zeta$ and $y = \texttt{abcdebcd}\mu\texttt{fi}$. Finally, one can select $\rho \in \Lambda^*$ to be equal to $\texttt{abcdebcdgfi}$. Note that $\rho \in L(S) \cap L(\omega)$.

that for every $X \subseteq \mathbb{A}$ it trivially holds that $c(y) = 0$, where y is an optimal alignment between $\eta|_X$ and $\hat{\sigma}$. Note that $rewrite(\eta|_{dom(h)}, h) = rewrite(\rho, h)$ and, thus, $\rho = \eta|_{dom(h)}$. Hence, it holds that $c(\gamma) = 0$.

($\Leftarrow$) Next, we demonstrate that if $c(\gamma) = 0$, then there exists $\eta \in L(\hat{S})$ such that $rewrite(\eta|_{dom(h)}, h) = (\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)|_{\mathbb{A}}$. Let $\hat{\sigma} := \pi_2(\gamma)|_{\hat{T}}$. It holds that $\hat{\sigma} \in \mathbb{E}_{\hat{S}}$, refer to Definition 3.6. Let $\eta := \hat{\lambda}(\hat{\sigma})|_{\hat{\Lambda}}$. If η contains more symbols from $dom(h)$ than ρ, then $\hat{\sigma}$ must be transformed by replacing some fresh transitions from $\hat{S}$ with transitions of S'' that they mimic such that the cost of an optimal alignment between ρ and $\hat{\sigma}$ is equal to zero and $\eta := \hat{\lambda}(\hat{\sigma})|_{\hat{\Lambda}}$ contains $|\rho|$ symbols from $dom(h)$; note that this transformation is always possible. It holds that $rewrite(\eta|_{dom(h)}, h) = (\langle \alpha \rangle \circ rewrite(\omega, f) \circ \langle \zeta \rangle)|_{\mathbb{A}}$ because, clearly, all the elements of ρ appear in η in the same order as they appear in ρ and, therefore, it holds that $rewrite(\eta|_{dom(h)}, h) = rewrite(\rho, h)$.

∎