Data & Knowledge Engineering 161 (2026) 102526

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

A framework for purpose-guided event logs generation

Andrea Burattin [a], Barbara Re [b], Lorenzo Rossi [b,*], Francesco Tiezzi [c]

[a] Software Systems Engineering, Technical University of Denmark, Denmark
[b] School of Science and Technology, University of Camerino, Italy
[c] Dip. Statistica Informatica Applicazioni, University of Florence, Italy

ARTICLE INFO

Keywords:
Process mining
Event log
Log generation
Simulation
Discovery
What-if analysis
Conformance checking

ABSTRACT

Process mining is a prominent discipline in business process management. It collects a variety of techniques for gathering information from event logs, each fulfilling a different mining purpose. Event logs are always necessary for assessing and validating mining techniques in relation to specific purposes. Unfortunately, event logs are hard to find and usually contain noise that can influence the validity of the results of a mining technique. In this paper, we propose a framework, named PURPLE, for generating, through business model simulation, event logs tailored for different mining purposes, i.e., discovery, what-if analysis, and conformance checking. It supports the simulation of models specified in different languages, by projecting their execution onto a common behavioral model, i.e., a labeled transition system. We present eleven instantiations of the framework implemented in a software tool by-product of this paper. The framework is validated against reference log generators through experiments on the purposes presented in the paper.

1. Introduction

Nowadays, process mining is an important discipline in extracting non-trivial information from the execution of business processes, thanks to the increasing usage of information systems that record event logs of the deployed processes [1]. The importance of process mining is also well recognized by companies, which appreciate the possibility of gathering knowledge from actual execution data and continuously improving their processes [2].

In brief, process mining is a collection of techniques to automatically extract information from event logs recorded during the execution of business processes. The effectiveness and the precision of process mining techniques are strictly related to the reliability of their mining algorithms, whose development requires validating them against different event logs [3], usually coupled with the models that generated them [4]. Mining algorithms extract different types of information according to the mining purpose they have to accomplish, e.g., process discovery, what-if analysis, and conformance checking. Therefore, to validate a process mining algorithm, it is important to use event logs that suit the purpose for which the algorithm has been devised [5]. For instance, given a family of discovery algorithms that leverages the same set of properties on the logs (e.g., the coverage of the direct following relations for the Alpha miner family [6]), a fair comparison of the algorithms would require logs that satisfy such properties. As stated in the literature [3,7,8], each mining purpose heavily relies on the quality, concerning specific properties, of the event logs given as input to the mining algorithms. Indeed, obtaining event logs fitting for a purpose is a complex, yet necessary, achievement [9].

Event logs are difficult to find, in particular, those directly extracted from deployed IT systems that refer to real-world installations [10]. Moreover, bad-quality logs hamper the use of process mining techniques, thus, researchers are encouraged to develop log generators that focus on a specific and explicit mining purpose [7]. This opens up the following research question.

* Corresponding author.
E-mail addresses: andbur@dtu.dk (A. Burattin), barbara.re@unicam.it (B. Re), lorenzo.rossi@unicam.it (L. Rossi), francesco.tiezzi@unifi.it (F. Tiezzi).

https://doi.org/10.1016/j.datak.2025.102526
Received 3 December 2024; Received in revised form 26 August 2025; Accepted 8 October 2025
Available online 15 October 2025

RQ1: *What are the relevant approaches in the literature for generating artificial event logs?*

In the literature, several approaches, e.g., [3,10–13], propose the automated generation of artificial event logs via the simulation of models in a predetermined language, e.g., BPMN or Petri Net. However, these are *purpose-agnostic*, thus not meant to produce event logs fulfilling properties required for a specific purpose. Instead, they simulate random execution traces, producing a different event log every time. This gap in the literature paves the way for the following research question.

RQ2: *How can event logs generators support different mining purposes?*

In this regard, we explore the possibility of driving the simulation to finally obtain specific execution traces that make the produced event log tailored to the mining purpose. To this aim, in this paper, we propose the PURPose-guided Log gEneration (PURPLE) framework. The main advantages of the PURPLE framework concerning existing simulators are as follows. PURPLE generates event logs specifically tailored to the purpose of the mining technique under investigation. To shape an event log, the framework performs a *guided simulation* of the input model that incrementally generates specific execution traces, until the desired purpose is satisfied. The simulation is guided by hints produced at each step based on the partial log generated up to that moment and the properties required for mining purposes. Additionally, the framework is meant to *simulate many kinds of business process models* (e.g., BPMN, Petri Net, WF-net) employing a common behavioral model that is the Labeled Transitions System (LTS). Besides the conceptual framework, we provide a tool, which implements eleven instantiations of the PURPLE framework that address process discovery, conformance checking, and what-if analysis mining purposes for BPMN processes and collaborations, and Petri-nets. To assess the impact of guided simulation in contrast to other available approaches, we formulate the following research question.

RQ3: *How does purpose-guided simulation perform in comparison to purpose-agnostic approaches?*

In this regard, we validate the advancements of our proposal to the state of the art on log generation, we carried out experiments measuring the quality of logs generated by PURPLE for the purposes it supports, and we compared these results with those of other log generators. Moreover, we assessed the performance of the tool implementation using different instantiations on models of increasing size and complexity.

Notably, this work is based on the paper *"A purpose-guided log generation framework"* [14], published in the proceedings of the 20th International Conference on Business Process Management. Besides describing our approach in greater detail, this article extends the scope of the original conference paper as reported below.

- We have revised the presentation of the framework by providing pseudocode descriptions of the main components.
- We have enriched the PURPLE framework by
  - a new component ensuring the termination of the log generation, namely a trace evaluator;
  - a new log evaluator regarding branching probabilities for what-if analysis, which has been implemented in the PURPLE tool as well as considered for the evaluation;
  - instantiations of the trace evaluator component dealing with trace length and cost;
  - a new semantic engine for BPMN collaboration models.
- We extended the validation to the new instantiations and we added performance analysis showing the execution times of PURPLE instantiations with models of increasing size.
- We have revised the related work section with a systematic scouting of the literature to include further approaches related to our proposal.

The rest of the paper is structured as follows. Section 2 provides a comprehensive review of the existing log generation approaches related to our work. Section 3 provides background notions on event logs and Labeled Transition Systems. Section 4 introduces the PURPLE framework and its components. Section 5 and Section 6 present some instantiations of the trace and log evaluator components of PURPLE respectively. Section 7 presents the PURPLE tool implementing the framework and its instantiations and reports the results of experiments comparing PURPLE with other reference simulators. Finally, Section 8 closes the paper by discussing assumptions, limitations, and opportunities for future work.

2. Related works

To answer **RQ1**, this section discusses the most relevant works on the generation of event logs from process model simulation. In describing them, we focus on features of interest for this paper, such as the generation of event logs tailored to a desired mining purpose from models specified in different modeling languages. Thus, we primarily focus on the types of event logs these approaches can generate and the models they support.

The literature review has been inspired by the methodology described in [15]. The review process comprises three phases: *planning* and *conducting* the literature review, and *reporting* on the review of the selected papers.

Table 1
Scopus query for the literature review.

```
TITLE-ABS((“event log*” OR “event data” OR “event stream*”) AND (“generation” OR “generator”
OR “simulator” OR “simulation” OR “synthesis” OR “synthetic”))
AND ( LIMIT-TO ( SUBJAREA , “COMP” ) )
AND ( LIMIT-TO ( DOCTYPE , “ar” ) OR LIMIT-TO ( DOCTYPE , “cp” ) )
AND ( LIMIT-TO ( LANGUAGE , “English” ) )
```

2.1. Planning and conducting

In the planning phase, we discussed defining the search query, the target papers' repository, and the inclusion and exclusion criteria. From the discussion, we decided to break down **RQ1** into two sub-questions related to the literature review:

RQ 1.1 *Which mining purposes are addressed by event log generation approaches?*

RQ 1.2 *Which input modeling languages are used to generate synthetic event logs?*.

Moreover, based on the research question, we defined a search query for retrieving relevant papers [16], see Table 1. As a digital library, we decided to focus on the *Scopus* library. We acknowledge that relying solely on Scopus represents a limitation to be considered when interpreting the comprehensiveness of the review. However, we adopt this choice since Scopus contains the highest number of unique articles [17], and indexes relevant works, discarding non-peer-reviewed articles [18]. As inclusion and exclusion criteria, we decided not to limit the results by publication year, while we limited the results to journal articles and conference papers written in English and belonging to the computer science field.

Then, we pass to the conducting phase that consists of the search for the research works and their analysis accordingly to the inclusion/exclusion criteria. Moreover, we apply backwards and forward snowballing to close the scouting [19,20]. From the application of the query, we obtained a first set of 1234 works. Then we scouted titles and abstracts of these works in order to exclude the ones that clearly are out of the scope of our analysis. This second run resulted in 54 papers, which have been read to assess their contribution. Finally, we identified 26 papers from the list, plus 3 from the snowball approach.

2.2. Reporting

The last phase consists of illustrating the data extracted during the review. First, we make some remarks on the identified research works. Of the 29 revised papers, 5 are journal papers, 21 are conference papers, and 3 (the ones from the snowballing) are not peer-reviewed. Moreover, two of them are our publications, i.e., the conference paper [14] and the demo paper [21], on top of which this paper builds on. Thus, we report on the remaining 27 papers reported in Table 2.

Referring to **RQ 1.1**, the selected papers propose approaches for event logs generation, both purpose-driven and purpose-agnostic; some of them provide tool support. As summarized in Table 2, 15 papers propose purpose-agnostic approaches that adopt random simulations. While the remaining 12 target seven classes of mining purposes, i.e., security, predictive process mining, generic benchmark of process mining techniques, compliance checking, drift detection, data-aware process mining, and multi-perspective process mining. Concerning **RQ 1.2**, we show that most of the approaches (i.e., 17) generate logs from the simulation of procedural models, i.e., Petri-nets, BPMN processes, Process trees, agents, and textual notations. The remaining approaches adopt declarative notations and process features.

Regarding purpose-agnostic approaches, Esgin and Karagoz present in [11] a solution to the problem of unlabeled event logs [44], proposing a synthetic event log generation approach. The generation of event logs can be tuned according to four parameters: the activity priority, an unexpected process termination probability, a noise threshold, and a branching probability for the choice gateways. Apart from these options, the simulation randomly executes the input Petri-net and cannot handle different mining purposes, nor have tool support.

Kataeva and Kalenkova propose in [12] grammar rules generating well-structured WF-Nets from which to produce logs. This work strongly limits the kind of logs that can be produced. Indeed, it handles just well-structured WF-Nets; moreover, logs cannot be tuned for specific mining purposes. Similarly, Burattin presents in [10] a tool called Process Log Generator (PLG2) that creates well-structured BPMN models and produces event logs from their simulation. To produce artificial models, PLG2 combines different control-flow patterns via context-free grammar according to options like the number of gateways or the presence of noise. Concerning PURPLE, this approach relies on random executions of the input model and works only with BPMN. Similarly, García-Bañuelos [23] and then de Medeiros and Günter [3] propose the CPN tool for the generation of event logs through the simulation of colored Petri-nets, subsequently enhanced with an IDE implementation by Verbeek and Fahland [24]. They point out the issues related to using real-life event logs to fine-tune mining algorithms and how the incompleteness of an event log or the presence of noise can compromise the evaluation of the mining algorithms. In the realm of Petri-nets, Kuhn et al. [25] introduce a methodology to simulate and generate event logs from data Petri-nets. These models are translated into the probabilistic programming language WebPPL and then used as simulation configurations. These approaches cannot tune the logs to produce since they rely on a random simulation of the input model.

Table 2
Summary of the literature review. In the "type" column, 'C' refers to conference; 'J' refers to journal; and 'N' refers to non-peer-reviewed papers.

Paper	Year	Type	Source	Purpose	Input Language
[22]	2024	C	Springer	Security	Agent-Based Modeling
[13]	2017	J	Elsevier	Agnostic	BPMN collaboration
[10]	2016	C	CEUR	Agnostic	BPMN process
[3]	2004	N	None	Agnostic	Colored Petri-net
[23]	2009	N	None	Agnostic	Colored Petri-net
[24]	2023	C	CEUR	Agnostic	Colored Petri-net
[25]	2024	C	CEUR	Agnostic	Data Petri-net
[26]	2023	C	Springer	Multi-perspective	Data Petri-net
[27]	2023	C	CEUR	Data	Data Petri-net
[28]	2016	C	CEUR	Agnostic	Declare
[29]	2018	C	CEUR	Agnostic	Declare
[30]	2015	C	Springer	Agnostic	Declare
[31]	2020	C	CEUR	Agnostic	Declare
[32]	2024	C	Springer	Agnostic	Declare
[33]	2023	C	CEUR	Agnostic	Declare
[11]	2019	C	Springer	Agnostic	Petri-net
[34]	2024	C	IEEE	Benchmark	Petri-net
[35]	2014	C	CEUR	Security	Petri-net
[36]	2013	C	Springer	Security	Petri-net
[37]	2024	J	SAGE	Predictive	Process configurations
[38]	2025	J	Elsevier	Agnostic	Process features
[39]	2024	C	Springer	Benchmark	Process features
[40]	2019	J	Springer	Benchmark	Process Tree
[41]	2016	C	CEUR	Benchmark	Process Tree
[42]	2022	C	CEUR	Concept drift	Process Tree
[43]	2020	J	Springer	compliance	SCIFF
[12]	2014	N	None	Agnostic	WF-net

Mitsyuk et al. face in [13] the problem of defining and generating logs from collaborative processes. They use an executable BPMN semantics supporting a subset of elements from the standard notation, such as tasks (also send/receive), sub-processes, parallel and exclusive gateways, and cancellation events. Moreover, they consider the data perspective, as data objects can store single data values used for driving exclusive choices. The result is a log generator integrated in the ProM framework that produces random event logs in *.xes* files. However, their approach only deals with a single modeling language and cannot be tuned for specific purposes.

The literature also provides papers focused on the generation of event logs for a specific mining purpose. Differently, Sommers et al. [34] define an approach for generating event logs with traces that deviate from the input model. The final aim is to have datasets for benchmarking process mining techniques. This approach simulates Petri-nets (or higher order Petri-nets) modified to produce traces accordingly to the deviations required by the user. With the same aim, Loreti et al. [43] define an approach for generating compliant and non-compliant traces from the simulation of procedural and declarative process models. However, these approaches are not meant to be applied to other mining purposes. Stocker and Accorsi introduce in [35,36] an approach for generating event logs for a specific purpose, i.e., testing security properties. They present a tool, called *SecSy*, that generates logs from the simulation of a Petri-net in a specific scenario. The simulation performs random execution of the model, then it applies transformations to the generated event log. These transformations remove or insert activities and modify traces to violate security properties. This approach takes care of one specific purpose for which the produced event logs are tuned, and of one modeling language (i.e., Petri-net). In the same field, Kamal et al. [22] define an approach to generate event logs with anonymized labels for addressing privacy concerns in the healthcare domain. Jouck and Depaire present in [40,41] a log generation approach specific for comparing discovery algorithms. They produce and then simulate a population of well-structured models from selected workflow patterns to ensure the presence of specific activity order relations chosen by the user. Grimm et al. [42] presents CDLG, a tool for generating logs with four types of concept drifts from the execution of process trees. These works use process trees to produce logs, and apart from process discovery and drift detection, further purposes are not considered. Mike Riess [37] introduces a framework for generating event logs tailored to predictive process monitoring experiments. The framework gets as input process configurations rather than a process model.

Finally, Grüger et al. present in [26] the SAMPLE approach for generating multi-perspective logs with realistic data. The approach semantically describes data via meta-models to ensure meaningful variable values and event logs that closely approximate reality. Moreover, the approach is implemented in the Data-Aware Event Log Generator [27], which enables users to generate synthetic event logs.

In particular, the literature presents approaches for log generation based on declarative process notations [28–31,41], which are out of our scope. The most significant attempts are by Alman et al. [32], which concentrate on generating event logs that come from the concurrent execution of a combination of declarative and procedural process models on the same case instances. The proposed approach produces an event log matching the concurrent execution of these hybrid models. More closely to a purpose-guided approach, Donatello et al. [33] propose a tool generating event logs that satisfy a given property.

Finally, the literature also proposes approaches that exploit machine learning techniques. Meneghello et al. [38] present the RIMS (Runtime Integration of Machine Learning and Simulation) approach to integrate predictions of deep learning models during the log generation to produce more accurate logs concerning the time perspective. Maldonado et al. [39] present GEDI, a framework for generating event data for benchmarking process mining. GEDI uses an unsupervised approach that creates event data according to user-selected features not directly related to a specific process model.

Summing up, the above-mentioned works mainly focus on generating random event logs without focusing on specific mining purposes. Moreover, they limit the simulation to single modeling languages and structured models. Lastly, some produce logs in non-standard formats, jeopardizing the compatibility with process mining tools.

3. Background notions

This section provides the notions we use in the rest of the paper. **Process models**, which graphically describe when and how the involved activities take place, are a common representation of process-aware systems in the process mining disciplines. Fig. 1(a) depicts a system modeled using the BPMN notation [45], which is one of the most used notations in the modeling practice [46]. In brief, a BPMN process model consists of a combination of different elements. *Events*, drawn as circles, represent the points from which the process starts (start events) or terminates (end events), though more complex usages are also allowed. *Activities*, drawn as rectangles with rounded corners, represent work units. *Gateways*, drawn as diamonds, act as either join nodes or split nodes. The gateway with the "+" symbol – AND gateway – enables parallel execution flows in the split mode and supports the synchronization of parallel flows in the join mode, while the one with the "×" symbol – XOR gateway – gives the possibility to describe choices in the split mode and pass-through elements in the join mode. *Sequence Edges*, drawn as solid connectors, specify the internal flow of the process. About its execution semantics, the model of Fig. 1(a) executes for first the activity A, then two options are possible: in the upper path the activities B and C are executed in parallel, while in the lower path, there is just the activity D. Then, before the termination of the process, activity E is executed.

An **event log** consists of a set of *cases*, each of which refers to some events that can be seen as one possible execution of a process. An *event* refers to the execution of a process activity, and it is described by a set of *attributes*. The most common attributes for a recorded event are the *timestamp* and *activity name*, but also other information can be captured, such as the *resource* involved in the activity execution or the monetary *cost* associated with it. The sequence of events related to a given case is called *trace*. Fig. 1(b) shows a table containing a fragment, with three cases, of an event log generated by the system modeled in Fig. 1(a). Each row in the table corresponds to an event occurred during the execution and contains information regarding the event attributes. For instance, the first three rows in Fig. 1(b) belong to *Case 1*, which contains three events. The first event reports the execution of an activity named A, which happened on May 5th, 2024 at 10:43 a.m.; the event was performed by the resource *Andrea* and cost 10.

Fig. 1(c) reports a **simple event log** [1, Ch. 5], which focuses only on the names of the executed activities. In this situation, an event log can be seen as a multiset of traces, where a *trace* is a sequence of activity names [47]. Worth noticing that different cases may have the same trace. In the considered example, the trace $\langle A, D, E\rangle$ occurs twice, while the trace $\langle A, B, C, E\rangle$ only once. The multiplicity of a trace is denoted in the simple event log by a positive integer (omitted when it is equal to 1).

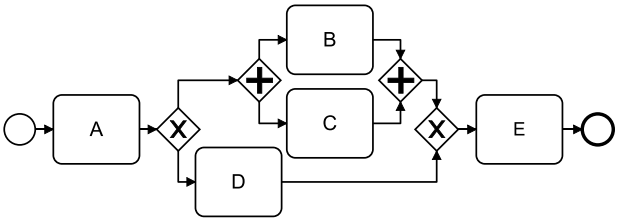
A way of generating event logs is through the simulation of a business process model [10]. The main idea is to repeatedly "execute" a model and to record, in a log file, all events observed during the execution. Simulators use the so-called *play-out engines*, like in [48,49], to execute models [1, Ch. 2]. An engine provides the moves a model can perform according to the semantics of the considered modeling language (e.g., the firing rule of Petri Nets [50] or the transition rules of BPMN operational semantics [51]), usually defined employing LTSs.

An **LTS** consists of *states*, representing the possible system configurations (i.e., the execution states of the model), and *labeled transitions*, corresponding to directed edges connecting states (representing moves in the model execution). Formally, a transition system is a triple $(S, L, \rightarrow)$ where: S, ranged over by s, is a set of states; $L = A \cup \{\tau\}$, ranged over by l, is the union of a set of (visible) activity labels A, ranged over by a, and a special label τ denoting an invisible activity; and $\rightarrow \subseteq S \times L \times S$ is a transition relation. The τ action is used to decorate those transitions of the LTS that do not refer to the performing of an activity included in the model but refer to the control of the execution flow, e.g., the execution of decisions, that can be neglected in the log generation.

Fig. 1(d) reports the LTS representing the behavior of the BPMN model in Fig. 1(a). This LTS is produced by the BPMN formal semantics described in [51]. Informally, the LTS is obtained as follows. Each configuration of the BPMN process model, i.e., each marking of tokens, corresponds to a state of the LTS. For example, the initial marking, where there is only one token placed on the start event, corresponds to the state s_i, while the marking obtained by one step of execution from the initial marking, where the token is moved to the sequence edge incoming into the activity A, corresponds to the state s_1. The execution of an activity of the BPMN model is rendered in the LTS in terms of a transition labeled by the name of the activity. For example, the execution of the activity A, which is enabled by the marking corresponding to the state s_1, is rendered by the transition connecting s_1 and s_2.

In an LTS, we call a state *initial* (resp. *final*) if it does not have incoming (resp. outgoing) transitions. The initial state, labeled s_i, corresponds to the initial configuration of the model, where its execution starts, while a final state, labeled s_f, is an ending configuration, which corresponds to a proper or an improper termination.

Finally, for a given LTS, $(S, L, \rightarrow)$ with $L = A \cup \{\tau\}$, it is possible to characterize: *sub-traces* as sequences of visible labels; *traces* as sequences of visible labels from the initial to a final state; and *logs* as multisets of *traces*. Formally, the sequence of labels $\langle a_1, a_2, \ldots, a_n\rangle$ with $a_1, a_2, \ldots, a_n \in A$ is a *sub-trace* if there exists $\langle l_1, l_2, \ldots, l_m\rangle$ with $l_1, l_2 \ldots, l_m \in L$ such that: *(i)* $\langle a_1, a_2, \ldots, a_n\rangle$ coincides with $\langle l_1, l_2, \ldots, l_m\rangle$ up to occurrences of τ; and *(ii)* $(s_1, l_1, s_2) \in \rightarrow$, $(s_2, l_2, s_3) \in \rightarrow$, $\ldots$, $(s_m, l_m, s_{m+1}) \in \rightarrow$ for some s_1, $s_2, \ldots, s_{m+1} \in S$. If s_1 is the initial state and s_{m+1} is a final state, the sub-trace is called *trace*.



(a) BPMN process model.

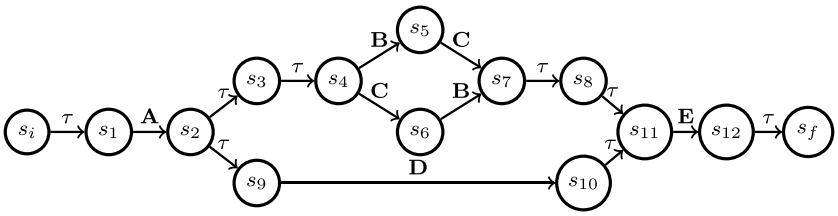
Case	Event Attributes				
	Activity	Timestamp	Resource	Cost	...
1	A	05/05/2024:10:43	Andrea	10	...
	D	05/05/2024:10:55	Barbara	20	...
	E	05/05/2024:11:03	Francesco	13	...
2	A	05/05/2024:10:53	Andrea	10	...
	B	05/05/2024:11:09	Lorenzo	5	...
	C	05/05/2024:11:21	Lorenzo	5	...
	E	05/05/2024:11:22	Francesco	15	...
3	A	05/05/2024:11:25	Andrea	10	...
	D	05/05/2024:11:31	Barbara	20	...
	E	05/05/2024:12:03	Francesco	13	...
...	...	...	...	...	...

..... *case* ---- *event* ____ *trace*

(b) Event log.

$$\{\langle A, D, E\rangle^2, \langle A, B, C, E\rangle\}$$

(c) Simple event log.



(d) Example of a labeled transition system.

Fig. 1. From process model to event log.

Traversing the LTS in Fig. 1(d) from the initial state s_i to the final state s_f, the sequences of visible labels associated with the transitions (thus, discarding the occurrences of τ labels) represent the execution traces that can be generated from the BPMN model in Fig. 1(a). Then, we obtain traces of the following forms: $\langle A, B, C, E\rangle$, $\langle A, C, B, E\rangle$, and $\langle A, D, E\rangle$. Multisets containing zero, one, or more occurrences of these traces, like the one in Fig. 1(c), are hence logs that can be generated from the BPMN model in Fig. 1(a).

4. PURPose-guided log gEneration framework

In this section and the next two, we present the PURPLE framework and its components. To answer **RQ2**, PURPLE is designed to simulate models for producing event logs with different properties that target different mining purposes. It supports the simulation of models specified with different languages by projecting their execution onto a common behavioral model, i.e., an LTS.

4.1. The PURPLE conceptual framework

The PURPLE conceptual framework is composed of the following components: a guided **simulator**, a **semantic engine**, a **trace evaluator**, and a **log evaluator**, see Fig. 2.

Starting from the **semantic engine**, this component is devoted to interpreting the input model representation according to its execution semantics and consequently to construct on the fly the corresponding LTS. The semantic engine starts creating an LTS

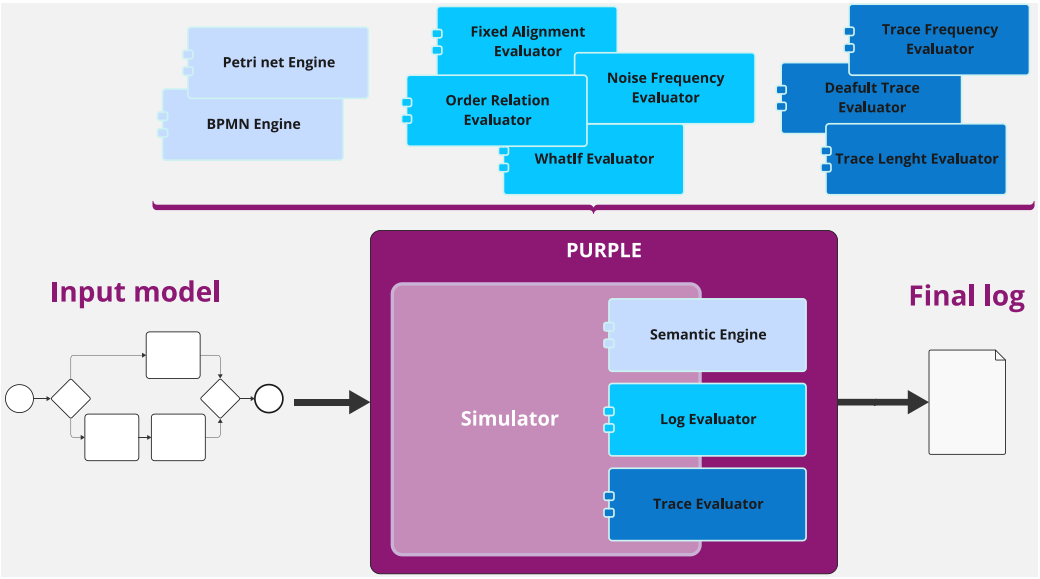


Fig. 2. PURPLE framework.

with only the initial state (i.e., the initial model configuration). Then, it offers the framework functions (i) to retrieve states of the LTS reachable by transitions with a specific label; (ii) to get states of the LTS reachable from a specific state by a sequence of transitions with specific labels; and (iii) to construct the LTS and get from the input model all the transitions outgoing a given state coupled with the relative target states. Relying on LTSs as behavioral representations of the input models, PURPLE can simulate any modeling language equipped with, or that can be mapped to, an executable semantics [1, Ch. 3]. This allows PURPLE to gain in terms of generalizability, since it can support any modeling language with a direct LTS semantics, e.g., [51,52], or adopting a mapping approach, e.g., [53,54]. It is worth noticing that PURPLE is not meant to support modeling languages with stochastic semantics that require other kinds of behavioral models, like Markov chains. On the other hand, the framework does not impose any constraint on the topology of the input models; ideally, it accepts models with arbitrary topology. Such restrictions, when needed, can be imposed by the semantic engine implementation, which can constrain specific topologies such as the absence of loops, safeness, or well-structuredness.

The **trace evaluator** component is responsible for deciding if to stop the generation of a trace during a simulation run. This component is twofold, it permits to specify conditions (i) related to the termination of a simulation run, e.g., a trace cannot be longer than 100 activities; and (ii) related to the mining purpose, e.g., the sum of the costs of the activities in a trace has to be lower than a threshold. Therefore, the trace evaluator permits adding to the log traces that do not necessarily end in the final state of the LTS. Hence, we refine the notion of trace, used throughout the rest of the paper, as follows: a sub-trace is a *trace* if it starts from the initial state and ends on a final state or evaluate holds.

The **log evaluator** component is responsible for evaluating an event log according to the peculiarities of the desired mining purpose. More practically, this component checks "how much" an event log satisfies the properties needed for the purpose under consideration. As a result, the log evaluator produces a *delta*. Formally, a delta is a set of sub-traces of the form $\{\langle l_{11}, l_{12}, \dots, l_{1i}\rangle, \langle l_{21}, l_{22}, \dots, l_{2j}\rangle, \dots, \langle l_{n1}, l_{n2}, \dots, l_{nk}\rangle\}$. The sub-traces in the delta drive the simulator to produce traces that contain them. They act as a bias indicating the parts of the LTS to be traversed, thus making the produced traces and the log suitable for the purpose. As each log evaluator is defined to deal with a specific mining purpose, the generated delta has to ensure the traces required by that purpose in the final event log. We clarify this point with a simplistic example that refers to Fig. 1, used just for the sake of presentation. Assuming a mining purpose that requires a log where model activities appear at least once, the log evaluator will select the activities not yet in the log and will produce a delta containing sub-traces of length one with the labels of the missing activities. For instance, assuming the log contains only case 2, Fig. 1(b), the only missing activity would be D. Thus, the log evaluator would produce a delta like $\{\langle D\rangle\}$. Consequently, this delta guides the simulator to produce a trace that passes from state s_9 in the LTS of Fig. 1(d). It is worth noticing that the effectiveness of the generated delta for the addressed mining purpose depends on the implementation of the log evaluator itself.

Finally, we introduce the **simulator**. This component produces specific traces from the execution of the input model. Its peculiarity lies in a *guided* traversal of the LTS to guarantee the production of traces, and hence a log, that satisfies the desired mining purpose. Indeed, differently from a purely random simulation, what the framework proposes is a **guided simulation** that takes as input from the log evaluator a delta that suggests execution paths, or part of them, to follow in the LTS traversal.

All components are coordinated by a looping routine, referred to as the PURPLE routine, which iteratively constructs an event log from a simulation of the input model. At each iteration, the delta generated by the log evaluator guides the simulator in the creation

of the next trace, which has to be accepted by the trace evaluator. Then, the routine terminates if the log evaluator does not produce further deltas. In the following subsection, we provide further details on the concrete implementation of the PURPLE routine and of the framework components.

4.2. The PURPLE framework instantiation

The conceptual framework can be instantiated using different implementations of the components presented so far. Except for the simulator that is fixed, the other components can be instantiated with (i) a semantic engine, to support a particular modeling language (e.g., BPMN, Petri Net); (ii) a trace evaluator, to specify the termination of a simulation run; and (iii) a log evaluator, tailored to a mining purpose (e.g., discovery, compliance checking, and what-if analysis). A framework instance is coordinated by the purpleRoutine function, see Listing 1. The routine requires a semantic engine, a log evaluator, a trace evaluator, and the simulator as input.

```
1 Log purpleRoutine(SemanticEngine se, LogEvaluator le, TraceEvaluator te, Simulator s):
2     Log l = new Log();
3     TraceSet delta = new TraceSet();
4     do:
5         l.add(s.globalSimulate(delta));
6         delta = le.evaluate(l);
7     while (delta != null)
8     return l;
```

Listing 1: purpleRoutine function of the PURPLE framework.

At the routine startup, the semantic engine creates an LTS containing only the initial state (i.e., the initial model configuration). The semantic engine offers the framework the functions find, to retrieve states of the LTS reachable by transitions with a specific label; getNexts, to get states of the LTS reachable from a specific state by a sequence of transitions with specific labels; and getMoves, to construct the LTS and get from the input model all the transitions outgoing a given state coupled with the relative target states. Notably, find and getNexts functions are the same in each semantic engine implementation; the actual difference between semantic engines lies in the implementation of the abstract method getMoves, which is responsible for interpreting the semantics of the input model and constructing the LTS.

Then, the PURPLE routine starts to simulate traces (line 5 of Listing 1) based on the guide it receives from the evaluation of the current log (line 6 of Listing 1). Once the generated log satisfies the purpose, i.e., the delta is null the routine terminates and the log is provided as output. We distinguish the case where the delta is null, indicating that the log generation has to be terminated, from the case where the delta is an empty set of traces, indicating that there is no guide for the simulation. More in detail, the generation of new traces is up to the function globalSimulate of the simulator, which takes as input a delta, see Listing 2.

```
1 TraceSet globalSimulate(TraceSet delta):
2     if (delta.isEmpty()):
3         return new TraceSet(randomSim());
4     else:
5         TraceSet traces = new TraceSet();
6         for st in delta:
7             traces.add(guidedSim(st));
8         return traces;
```

Listing 2: globalSimulate function of the Simulator component.

In case the delta is empty (line 2 of Listing 2), for instance, when the log evaluator has not performed any comparison yet, the simulator randomly executes the model via the randomSim function. The randomSim function generates a trace from a random path in the LTS from the initial state until it reaches a final state or the trace evaluator evaluates the trace as true. Differently, in case the delta contains one or more sub-traces (line 4 to 8 of Listing 2), the simulator performs a guided simulation for each of these sub-traces *st*, considering them as breadcrumbs to follow for logging a specific trace in the LTS.

The guidedSim function, see Listing 3, looks for a path in the LTS corresponding to the sub-trace in input *st* and finalizes it by adding a prefix, i.e., a sub-trace from the initial state to the state where *st* begins; and a suffix, i.e., a sub-trace from the state where *st* ends and a state that is final or if the trace evaluator returns true.

```
1  Trace guidedSim(Trace st):
2      StateSet states = se.find(st.first());
3      Trace st_p = new Trace();
4      st_p.add(st.first());
5      st.removeFirst();
6      for (s in states):
7          if addPrefix(st_p, s):
8              if recursiveSim(st_p, s, st):
9                  return st_p;
10     return new Trace();
```

Listing 3: guidedSim function of the Simulator component.

More in detail, the guided simulation starts looking in the LTS for paths that start with the first element in the sub-trace st, i.e., st.first(). This activity is demanded to the function find of the semantic engine (line 2 of Listing 3). This function, called on st.first() returns the states that are the target of transitions labeled with st.first(). Then, the function initializes the construction of the sub-trace st_p used to store the trace simulated up to that moment. The first element st.first() is indeed added to st_p and removed from st (lines 4 and 5 of Listing 3). At this point, the function explores the possibility of continuing the guided simulation starting from each state found so far. If no states are found, the function returns an empty trace (line 10 of Listing 3), otherwise, the function tries to go ahead with one of the found states s (line 6 of Listing 3). The simulator adds to st_p a prefix sub-trace that leads to s using function addPrefix (line 7 of Listing 3).

Then, it calls the recursive function recursiveSim, see Listing 4. The function takes as input the trace under construction st_p, the state of the LTS s_{curr} from which it continues the simulation, and the remaining part of the hint of the delta, i.e., the sub-trace st. This function tries to complete the trace st_p with the labels corresponding to the remaining part of st.

```
1  boolean recursiveSim(Trace st_p, State s_curr, Trace st):
2      if (st.isEmpty()):
3          return addSuffix(st_p, s_curr);
4      else :
5          Label l = st.first();
6          StateSet states = se.getNexts(s_curr,l);
7          for (s_next in states):
8              st_p.add(l);
9              if (te.evaluate(st_p)):
10                 return false;
11             st.removeFirst()
12             if (recursiveSim(st_p, s_next, st)):
13                 return true;
14             else:
15                 st_p.removeLast();
16                 st.add(l);
17         return false;
```

Listing 4: recursiveSim function of the Simulator component.

In brief, recursiveSim unwinds st until it does not contain labels anymore (base case of the recursion, line 2 of Listing 4). In this case, the function addSuffix (line 3 of Listing 4) adds to st_p a suffix sub-trace that starts from s_{curr} and reaches a final state of the LTS or a state in which the trace evaluator evaluates it as true. In the cases where st still contains labels, the recursiveSim calls function getNexts of the semantic engine to get the states reachable from s_{curr} by a transition with a visible label l, or by a sequence of transitions with invisible labels that includes a visible label l (lines 5 and 6 of Listing 4). For the sake of presentation, function getNexts is described in detail in Appendix.

Back to the recursiveSim, if the set of next states $states$ is empty, the function returns false (line 16 of Listing 4). Otherwise, the recursiveSim tries to continue the guided simulation from one of the found state s_{nexts} (line 6 to 15 of Listing 4). More in detail, st_p is increased with l and is evaluated again by the trace evaluator. If the trace evaluator asses that st_p is a completed trace, the recursiveSim returns $false$ to the previous recursive step. Otherwise, the hint st is decreased of the first label (line 10 of Listing 4) and another recursive step is invoked using as parameter the current trace st_p, the chosen state s_{next}, and the remaining part of the hint st.

5. Trace evaluator instantiations

To better show the role of the trace evaluator, we present three possible instantiations of the component, such as default, trace length, and trace cost evaluators, and discuss how a new one can be created.

Default trace evaluator. The default trace evaluator (Listing 11 in Appendix) is the simplest instantiation of this component. Essentially, it always evaluates the current trace as false. Even if its functioning is trivial, this trace evaluator allows the simulator to generate traces that only end in a final state.

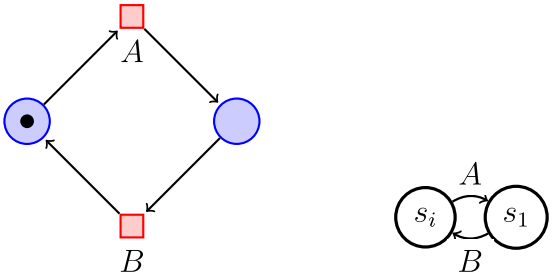
Trace length evaluator. The default trace evaluator suffers when the input model can generate traces of infinite length due to loops or unsafeness. This problem can be avoided using a trace evaluator that bounds the length of the traces. This is the trace length evaluator (Listing 12 in Appendix), which can be instantiated by giving as input the desired threshold for the maximum trace length. Consequently, the evaluate function returns true when the generated trace has overcome the threshold.

To show the relevance of this evaluator, let us consider the Petri Net models in Fig. 3. The semantics of the Petri Net in Fig. 3(a) is given by the LTS in Fig. 3(b), which has finite states but not a final one. Instead, the semantics of the Petri Net in Fig. 3(b) is given by the LTS in Fig. 3(d), which has infinite states (and no final state). In both cases, the default trace evaluator does not work well because it allows the simulation to diverge. Instead, using the trace length evaluator, the simulation can be stopped when a given number of states is reached.

Trace cost evaluator. The trace cost evaluator (Listing 13 in Appendix) is an example of a trace evaluator that predicates over event attributes. The goal is to limit the cost of a trace, i.e., the sum of the costs of the events composing it, to a threshold. Similarly to the previous trace evaluator, this one is instantiated with the maximum cost allowed for a trace. Then the evaluate function assumes that the events have an attribute named *cost* and checks if the sum of the costs of the events in a trace is greater than the threshold.

(a) (b)

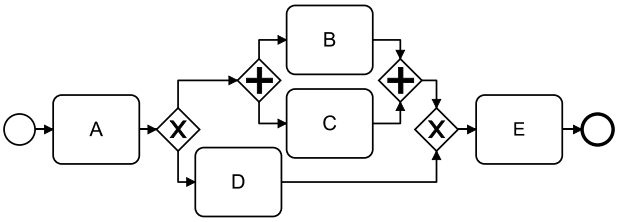
A Petri Net model of a token generator (a) and the corresponding LTS (b).



(c) (d)

A Petri Net model containing a loop (c) and the corresponding LTS (d).

Fig. 3. Examples of Petri Nets and their corresponding LTSs.



(a)

	A	B	C	D	E
A	#	→	→	→	#
B	←	#	||	#	→
C	←	||	#	#	→
D	←	#	#	#	→
E	#	←	←	←	#

→: sequential, ||: parallel, #: exclusive

(b)

Fig. 4. An input process model (a), and the related footprint matrix (b).

Custom trace evaluator. The above trace evaluator definitions are only examples showing how to impose simple constraints on the traces the simulator can generate. More complex trace evaluators can be developed from scratch or by combining different evaluators. For example, we can construct a trace evaluator that avoids the generation of traces over 100 events or that costs more than 100€ by combining the trace length and the trace cost evaluators (Listing 14 in Appendix).

6. Log evaluator instantiations

We present here five instantiations of the log evaluator tailored to the three purposes addressed by PURPLE such as process discovery via order relations and via frequencies, what-if analysis via branching probabilities, and conformance checking via noise frequencies and via fixed align cost.

6.1. Process discovery in PURPLE

The first instantiations of the log evaluator that we consider regard the process discovery. To check the reliability of a discovery algorithm, or to conduct a benchmark of different techniques, logs presenting specific characteristics are required. The evaluators address two specific discovery purposes: one is devised for algorithms relying on the order relation between activities, such as the Alpha algorithm [6], while the other one is for algorithms relying on frequencies, such as the Heuristics miner [55].

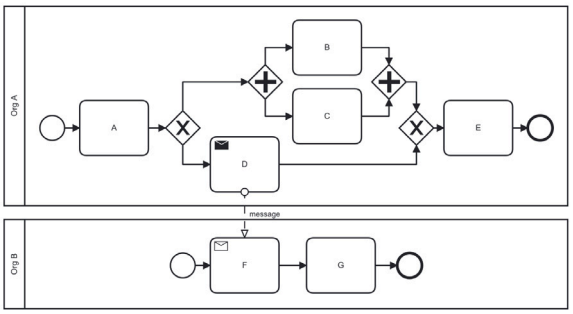
Process discovery via order relations. This log evaluator aims to make the simulator generate event logs for discovery algorithms that build the output models on the basis of the order relations between activities. These algorithms, e.g., the Alpha family, scan the input event log to find the *footprint* matrix of the original model. Assuming that an activity Y directly follows an activity X ($X > Y$) if and only if there exists a trace in the log where Y appears immediately after X, the footprint matrix can contain three kinds of order relations [6, Def. 3.2]. The sequence relation, denoted by $X \rightarrow Y$, holds if and only if $X > Y$ and $Y \not> X$. The parallel relation, denoted by $X \parallel Y$, means that X directly follows Y and vice versa ($X \parallel Y \iff X > Y$ and $Y > X$). The last relation, denoted by $X\#Y$, is used when two activities are unrelated, i.e., neither X directly follows Y nor Y directly follows X ($X\#Y \iff X \not> Y$ and $Y \not> X$). Considering the model in Fig. 4(a), the corresponding matrix is provided in Fig. 4(b). Depending on the input model, the generation of the footprint matrix may slightly differ. In the case of a collaboration, relations between sending and receiving activities are also included in the footprint matrix. Fig. 5 shows an example of a collaboration diagram and the related footprint matrix; in addition to the relation between activities of the same process, the matrix includes a sequential relation between the send task D and the receive task F. To obtain an accurate version of the original model, the input event log has to provide as many order relations as possible to fill the footprint matrix. For instance, logging multiple times the same trace is useless, as it always provides the same order relations. This can be achieved through a log evaluator that guides the simulation into the discovery of the footprint matrix, avoiding producing duplicates of the same trace, thus generating the smallest log covering the relations in the footprint matrix.

```
1  float coverage;
2  RelationsMatrix footprint;
3
4  OrderRelationEvaluator(float c, Model m):
5      coverage = c;
6      footprint = calculateFootprint(m);
7
8  TraceSet evaluate(Log log):
9      TraceSet delta = new TraceSet();
10     if log.length() = 0;
11         return delta;
12     RelationsMatrix currentFootprint = calculateFootprint(log);
13     RelationSet missingRelations = compare(footprint, currentFootprint);
14     if (missingRelations.size() / footprint.size() ≥ coverage):
15         return null;
16     for (Relation r in missingRelations):
17         if (r.type() = "→"):
18             delta = delta.add(⟨r.first(),r.second()⟩);
19         else if (r.type() = "||"):
20             delta = delta.add(⟨r.first(),r.second()⟩);
21             delta = delta.add(⟨r.second(),r.first()⟩);
22         else if (r.type() = "#"):
23             delta = delta.add(⟨r.first()⟩);
24             delta = delta.add(⟨r.second()⟩);
25     return delta;
```

Listing 5: Evaluator for process discovery via order relations

Listing 5 provides the pseudocode of the log evaluator for process discovery via order relations. The constructor takes as input (line 4) the coverage, i.e., a number from 0 to 1 indicating the percentage of order relation to be *covered* in the final log, and the input model from which to calculate the *footprint* matrix. While the evaluate function takes as input the current log. In case the log does not contain traces yet (line 10), the function returns an empty set as delta, triggering a random simulation. Otherwise, the function calculates the footprint matrix of the current log (line 12), and compares it to the one of the model to retrieve only the relations that are still missing (line 13). Therefore, if the number of missing relations divided by the number of relations in the model exceeds the required coverage, the function returns and stops the log generation (line 15). While, if it is not the case, each missing relation is analyzed separately (lines 16–24). Depending on the type of relation (sequence, parallel, or choice) (lines 17,19,22) the delta is increased with traces composed of the activities involved in the missing relation. Specifically, if it is a sequence relation, the resulting trace contains the first member of the relation followed by the second (line 18). If it is a parallel relation, the function adds in the delta two traces with the activities in the relation in any order (lines 20 and 21). Finally, if it is a choice relation, the delta is increased by 2 traces, each of which contains an event labeled with the activities involved in the relation (lines 23 and 24). The resulting delta is therefore returned to the simulator (line 25).

Considering the example in Fig. 4, the first time the evaluate function is invoked, the log is empty and it returns an empty *delta*. This leads to a random simulation of the model (see lines 2–3 of Listing 2). Supposing that the first simulation run is $\langle A, B, C, E\rangle$, the simulator performed tasks A, B, C and E, one after the other. Thus, the semantic engine can add to the LTS (containing only the initial state) the discovered states and transitions, producing the LTS in Fig. 6(a) to the exclusion of dotted states and transitions that are still to be discovered. Moreover, the discovered trace is added in the empty log, resulting in Fig. 6(b). Notably, to speed up the generation of the entire LTS, the semantic engine adds all the discovered states, even if they do not take part in the produced trace (see states s_6 and s_9). In the second run, the log evaluator calculates the order relations by considering the current log. The log identifies 3 order relations: $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow E$; the other activities are still unrelated, thus the resulting footprint matrix is the one in Fig. 6(c). At this point, the log evaluator compares the obtained footprint matrix with the one of the original model (Fig. 4(b)) to calculate the missing relations and produces the delta for the upcoming simulation step. The order relations

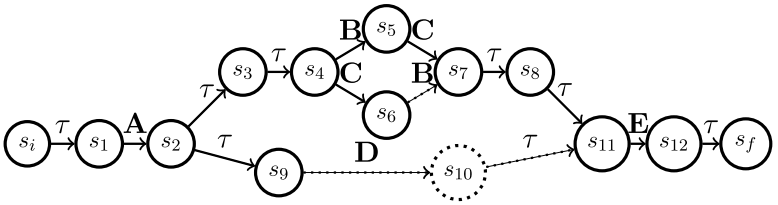


	A	B	C	D	E	F	G
A	#	→	→	→	#	#	#
B	←	#	||	#	→	#	#
C	←	||	#	#	→	#	#
D	←	#	#	#	→	→	#
E	#	←	←	←	#	#	#
F	#	#	#	←	#	#	→
G	#	#	#	#	#	←	#

→: sequential, ||: parallel, #: exclusive

(a) (b)

Fig. 5. An input collaboration model (a), and the related footprint matrix (b).



(a)

$\{\langle A, B, C, E\rangle\}$

(b)

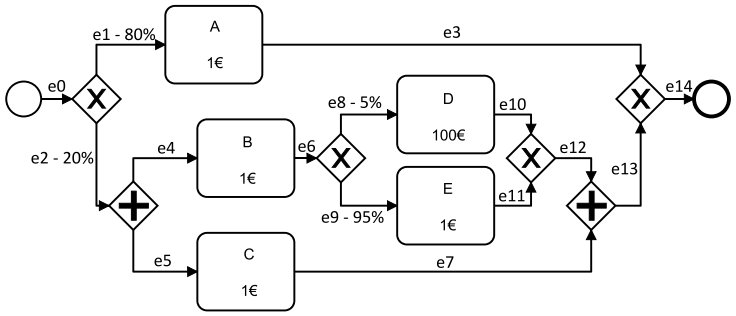
	A	B	C	D	E
A	#	→	#	#	#
B	←	#	→	#	#
C	#	←	#	#	→
D	#	#	#	#	#
E	#	#	←	#	#

(c)

Fig. 6. LTS (a), log (b), and footprint (c) resulting from the first run of simulation.

that are still missing are: $A \rightarrow D$, $B \rightarrow E$, $C \rightarrow B$, and $D \rightarrow E$. These relations are translated into sub-traces composing the delta as following:$\{\langle A, D\rangle, \langle B, E\rangle, \langle C, B\rangle, \langle D, E\rangle\}$. Since the delta is not empty, this time is crucial to guide the simulator to find additional traces containing the missing relations; in doing that, the simulator also relies on the current LTS. Considering the first hint of the delta, $\langle A, D\rangle$, the simulator looks for a state with an incoming transition labeled by A, that is state s_2, then it goes forward in the LTS to find a transition labeled by D. Being s_3 already visited, the simulator goes ahead to state s_9 that corresponds to a state in which activity D is enabled. Then, the simulator finalizes the trace until it reaches a final state, logging the trace $\langle A, D, E\rangle$. Instead, considering the hint $\langle C, B\rangle$ of the delta, the simulator has two states with an incoming transition labeled by C, i.e., s_6 and s_7, from which it starts looking for a transition labeled by B. State s_7 leads only to a transition labeled by E, while state s_6 leads to state s_7 with a transition labeled by B. Thus, the simulator follows this latter path in the LTS, logging the trace $\langle A, C, B, E\rangle$. After the second simulation run, the LTS produced by the simulator corresponds to the one in Fig. 6(a) considering also dotted states and transitions. The resulting log is $\{\langle A, B, C, E\rangle, \langle A, C, B, E\rangle, \langle A, D, E\rangle\}$. The log evaluator takes this log as input and assesses that all relations in the footprint matrix are covered, i.e. 100% of completeness is achieved. Notably, in this example, we required the highest level of completeness, but the user could specify a lower threshold.

Process discovery via frequencies. This log evaluator aims at making the simulator generate event logs for discovery algorithms based on frequencies. For instance, the Heuristics algorithm relies on threshold values for filtering less frequent behaviors, e.g., the occurrences of an activity or an order relation. To this aim, this instantiation of the log evaluator permits the choice of trace frequency. The resulting event log can be tuned to represent more realistic situations where behaviors could be less or more frequent than others. Logs of that form suite to compare the filtering approaches of different algorithms. To address this purpose, the log evaluator extracts the set of traces the model can perform and information regarding the loops. Then, the user specifies the



Activity	Occurrence
A	80%
B	20%
C	20%
D	1%
E	19%

(b)

Fig. 7. An input process model including scenario parameters (a), and its activity occurrence rates (b).

percentage of occurrence for each trace, a threshold value for the maximum number of repetitions of loops, and a minimum number of traces to be produced. Notably, in case the input model is a collaboration of processes, the log evaluator extracts groups of traces belonging to the same collaboration case, then a single percentage is associated with each group.

Therefore, during the log generation, the log evaluator implemented for this purpose compares the occurrences of traces and the thresholds for the loops chosen by the user with the current log and generates a delta accordingly. If some of these values are lower than requested, the log evaluator gives the simulator a delta containing the entire traces still infrequent in the log. If a trace includes a loop, the log evaluator modifies the trace in the delta by repeating the loop (for a random number of times below the given threshold). Then, the delta, which contains only complete execution traces of the input model, guides the simulator from the initial to the final state of the LTS. Once the minimum number of traces in the log is reached, and the requested occurrence percentages are satisfied, the log evaluator stops the simulation.

```
1  TraceFrequencyEvaluator(int minT, TraceSet traces, FreqMap freq, int loopRep):
2      ...
3
4  TraceSet evaluate(Log log):
5      TraceSet delta = new TraceSet();
6      for (Trace t in traces):
7          currentFreq = getFreq(t,log);
8          tFreq = freq.get(t);
9          if (currentFreq < tFreq):
10             if (hasLoop(t)):
11                 t = repeatLoop(t, loopRep);
12             delta = delta.add(t);
13     if (delta.isEmpty() & log.length() > minT):
14         return null;
15     return delta;
```

Listing 6: Evaluator for process discovery via frequencies.

Listing 6 provides the pseudocode of the evaluator implemented in PURPLE for the process discovery via frequencies. The evaluator is instantiated by giving as input a minimum number of traces to generate, the set of traces that the input model can generate, and the user selections for trace frequencies and loop repetitions (line 1). For the sake of readability, we omitted the entire implementation of the constructor since it assigns only input parameters to class variables as for the previous evaluator. The evaluate function iterates over the traces the model can generate (line 6) and checks if its frequency in the current log (line 7) is lower than requested (line 9). In this case, if it contains a loop, the trace is modified by repeating the loop a number of times lower than requested (line 11) and then added to the delta (line 12). Then, if the resulting delta contains traces it is returned to the simulator (line 15), while in case the delta is empty (since trace frequencies are sufficient) and the log size is greater than requested (line 13) the function returns and stops the log generation.

6.2. What-If analysis via branching probabilities

This log evaluator is tailored to *what-if analysis via branching probabilities*. In this case, the generated event logs will be exploited for the analysis of quantitative aspects of model execution, e.g., the cost and the completion time of the model activities, the resources performing them, and the branching probability. This permits us to answer questions and make predictions without data from a real environment [56]. It is worth noticing that this evaluator addresses a limited set of simulation features needed for an extensive what-if analysis. More complex instantiations could add activity duration, resource schedule, and other features.

Considering the BPMN model in Fig. 7(a), a typical question in the what-if analysis is *"what is the average cost of the model execution if we consider the showed scenario?"*. The scenario we are considering reports the cost of the activities, that is 1€ for each

of them, apart from activity D which costs 100€ and imposes the branching probability. This information is crucial for tuning the log the PURPLE tool has to produce. Indeed, it constrains the occurrence rate of an activity, as reported in the table of Fig. 7(b). The occurrence rate is calculated as follows. Being the sequence flow *e1* activated the 80% of the times the process executes, the following activity A will be performed the 80% of the times as well. Considering activities B and C, the percentage decreases to 20% due to the probability of choosing the sequence flow *e2*. In this 20% of times, one activity between D and E will run depending on the probabilities associated with sequence flows *e8* and *e9*. More precisely, activity D is executed the 5% of the times the process follows the sequence flow *e2*, that corresponds to the 1% of the entire process executions; activity E the 19%. To get reliable results from the analysis, the occurrence rate of activities in the event log has to correspond to the ones discussed above.

```
1  BranchingProbabilitiesEvaluator(int numT, Model model, CostMap costs, float precision):
2      occurr = getOccurrencies(model);
3      ...
4
5  TraceSet evaluate(Log log):
6      TraceSet delta = new TraceSet();
7      for (Activity act in model):
8          if (|occurrence(act,log) / occurr.get(act)| ≤ precision):
9              delta.add(<act>);
10     if (delta.isEmpty()):
11         return null;
12     return delta;
```

Listing 7: Evaluator for branching probabilities.

Listing 7 shows the implementation of the log evaluator used for this purpose. It is instantiated by specifying the minimum number of traces to generate, the input model, the cost of each activity, and the precision in reproducing the exact rate of occurrence of the activities. The evaluate function has the duty of producing an event log where the activities appear according to the percentages of activity occurrences induced by the scenario, see Fig. 7(b). Therefore, for each activity in the model (line 7), it compares the occurrence percentages of the model activities in the current log with the desired occurrences (line 8). In case the activity occurrence is lower than expected, the evaluator adds to the delta a new sub-trace with just the considered activity (line 9). Only in case all activities reach the desired occurrence, the delta remains empty, and the evaluator returns null (lines 10–11).

For instance, considering the example, if the first produced trace is $\langle B, C, D\rangle$, the evaluator calculates the activity percentages as follows: $A = 0\%$, $B = 100\%$, $C = 100\%$, $D = 100\%$, $E = 0\%$. This means that activities B, C, and D appear more than expected, while A and E are not sufficiently present in the log. For this reason, the delta for the next simulation step will include exactly the sub-traces $\langle A\rangle$ and $\langle E\rangle$. These sub-traces in the delta guide the next simulations to log traces including them. Only when each activity has reached the desired percentage of occurrence, the generation of the log can terminate. The resulting event log is $\{\langle A\rangle^{160}, \langle B, C, E\rangle^{38}, \langle B, C, D\rangle^{2}\}$. It contains 200 traces, where activity A appears 160 times, B and C 40 times, E 38 times, and D 2 times; that is exactly the result required up to the 100% of precision.

6.3. Conformance checking in PURPLE

Lastly, we present two log evaluators related to conformance checking, a family of techniques for comparing a model and a log. In particular, we consider techniques based on [1]. They permit to spot differences between the expectation (i.e., the process model) and the reality (i.e., the event log). Alignments explicitly show where deviations are located and which activities are involved. Computing alignments is an expensive task, especially in the presence of models with huge state-space, and there exist different approaches implementing it [57]. To check the reliability of such techniques, or to compare their performances, it is necessary to have logs embedding traces with deviations from the normal behavior, i.e., noisy behaviors. To this end, we propose two instantiations of PURPLE producing event logs from BPMN and Petri-nets with a precise amount of noisy behavior, or with a precise alignment cost.

Conformance checking via noise frequencies. This log evaluator generates event logs with the desired percentages of noisy traces. The literature identifies types of noise that can affect a trace in an event log [58]; here we consider the following: *missing head*, a trace without some of the initial events; *missing tail*, a trace without some of the final events; *missing episode*, a trace without some of the intermediate events; *order perturbation*, a trace where some events appear in a wrong order; and *additional event*, a trace in which appears an alien event. This log evaluator takes as inputs a model to simulate, a number of traces to generate, a percentage of occurrence for each type of noise, and a precision in reproducing the noise percentages. Whenever it is invoked, the log evaluator sends an empty delta to make the simulator produce a random trace without noise. Then, the log evaluator compares the percentage of occurrences for each type of noise in the current log with respect to the requested one. The trace is hence modified introducing the type of noise farthest from the requested occurrence. In case of *missing head*, *missing tail*, or *missing episode*, PURPLE removes a random number of events from the head, from the middle, or from the tail of the trace, respectively. In the case of *order perturbation*, it swaps two or more events in the trace, while in the case of *additional event* it inserts an event named differently from every activity name in the model. Once the log evaluator finds the desired noise percentages and number of traces, it returns the final log.

Listing 8 provides the pseudocode of the log evaluator for conformance checking via noise frequencies. It takes as input the number of traces to generate, the frequencies of each type of noise, and a precision value from 0 to 1 (line 1). The evaluate function checks if the current log is empty, in this case, it returns an empty delta to get back a random trace (lines 5–7). Otherwise, the function calculates the current noise frequencies from the current log, and checks which of these frequencies is the lowest with

respect to the user choice (lines 8–15). In doing so, the function saves temporally in the variable *minimum* the lowest frequency, and in *noiseType* the corresponding type of noise. Then, the function checks if the minimum frequency reaches the requested precision and if the log size reaches the one requested (line 16). In this case, the function returns and stops the log generation. Otherwise, the evaluator retrieves from the log the last trace generated and adds to it the type of noise corresponding to the minimum frequency (line 18).

```
1  NoiseFreqEvaluator(int numT, FreqMap noiseFreq, float precision):
2      ...
3
4  TraceSet evaluate(Log log):
5      TraceSet delta = new TraceSet();
6      if (log.length() = 0 ):
7          return delta;
8      currentFrequencies = calculateNoise(log);
9      float minimum = 1;
10     String noiseType = null;
11     for (String noise in noiseTypes):
12         currentPrecision = currentFrequencies.get(noise) / noiseFreq.get(noise) ;
13         if (currentPrecision < minimum):
14             minimum = currentPrecision;
15             noiseType = noise;
16     if (minimum = 1 − precision&log.length() ≥ numT):
17         return null;
18     addNoise(getLastTrace(log), noiseType);
19     return delta;
```

Listing 8: Evaluator for conformance checking via noise frequencies.

Conformance checking via fixed align cost. This log evaluator aims at generating event logs with a precise amount of noise that involves a specific cost for the alignment. Roughly speaking, the alignment cost indicates the number of deviations between the model and the log. An alignment cost equal to zero indicates a perfect match between the log and the model, while higher costs indicate the presence of non-compliant behaviors. Synchronous moves between trace and model cost zero, while moves that can be performed only in the model or only in the trace usually cost 1. The same trace can be aligned to the model following different execution paths and leading to different costs; the one to consider for calculating the mean value is the lowest i.e., the optimal alignment. The overall alignment cost is the average of the optimal alignments for each trace in the log. Considering the model in Fig. 4(a), a noisy trace could be $\langle B, C, E\rangle$, where the event labeled with A lacks. By aligning this trace through the path $\langle A, D, E\rangle$, only the last event matches, thus we have to perform two moves in the trace and two moves in the model that cost in total 4. While following the path $\langle A, C, B, E\rangle$ and $\langle A, B, C, E\rangle$, the alignment costs are respectively 3 and 1. Therefore, the optimal alignment cost to consider is the lowest one, i.e. 1.

Here, the log evaluator takes as input a model, a desired alignment cost, a log size, and a precision in reproducing the exact alignment cost. Before evaluating the current log, it extracts the set of traces that the model can produce and uses them later for calculating the alignment costs. Then, similarly to the previous purpose, the log evaluator receives from the simulator traces without noise, perturbs them with a type of noise, and updates the reached alignment cost. Every time a noisy trace is added to the current log, the log evaluator calculates the optimal alignment cost computing the minimum among the Levenshtein distances [59] between the noisy trace and traces previously extracted from the model.

```
1  FixedAlignEvaluator(int numT, float cost, TraceSet traces, float precision):
2      ...
3
4  TraceSet evaluate(Log log):
5      if ( calculateCost(log,traces) / cost > precision&log.length() > minT):
6          return null;
7      TraceSet delta = new TraceSet();
8      addRandomNoise(getLastTrace(log));
9      return delta;
```

Listing 9: Evaluator for conformance checking via fixed align cost.

Listing 9 provides the pseudocode of the log evaluator for conformance checking via fixed align cost. It takes as input the number of traces to produce, the alignment cost required, the set of traces the model can produce, and the precision in reaching the exact alignment cost (line 1). The evaluate function calculates the cost for aligning the current log with the traces the model can perform (via function CALCULATECOST) and compares it with the cost to reach considering the precision value. If the reached cost is sufficiently high and the log size exceeds the required dimension (line 5), the function returns and stops the log generation (line 6). Otherwise, it adds a random type of noise to the last trace added in the log (line 8).

7. Validation

This section presents a list of experiments on several instantiations of the PURPLE framework using the PURPLE tool we developed. The final aim of this section is to answer **RQ3**, thus evaluating if the event logs generated by PURPLE better satisfy the selected mining purpose and what is the impact of implementing guided simulations rather than purpose-agnostic approaches.

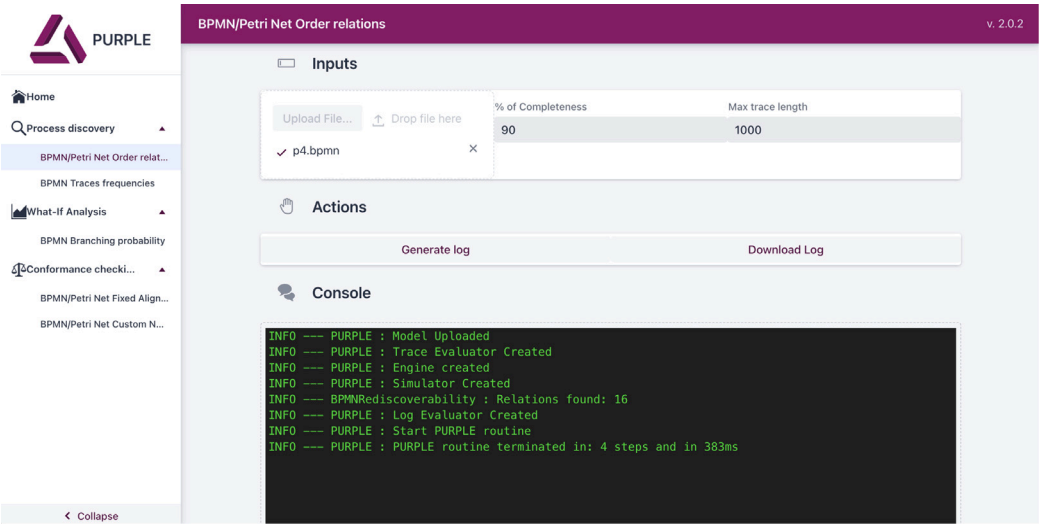


Fig. 8. The PURPLE tool interface.

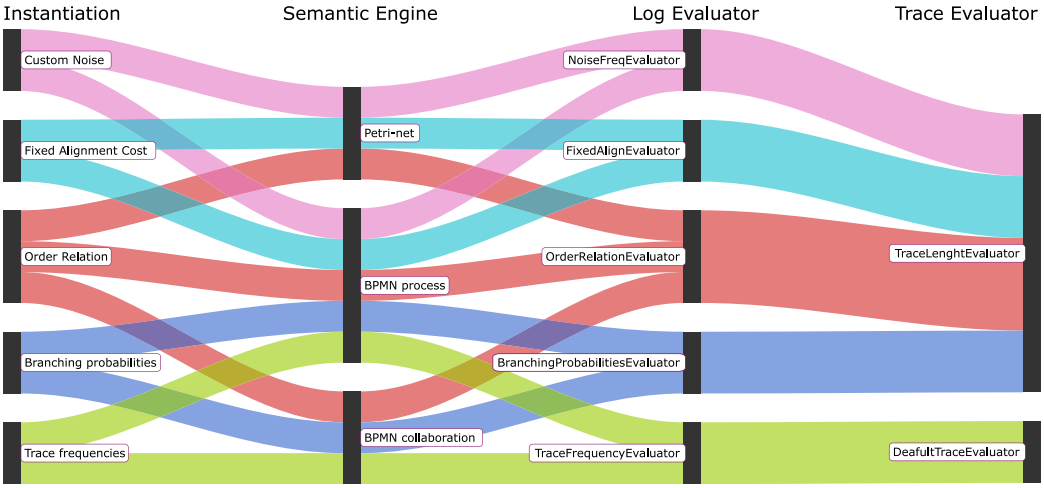


Fig. 9. Instantiations of PURPLE.

7.1. Setup

The PURPLE tool, Fig. 8 is a progressive web application implementing the homonym framework and its instantiations. It can be used online as a service or executed on a local machine. Tool, source code, instructions, and examples are available at https://pros.unicam.it/purple. The PURPLE tool implements eleven instantiations of the framework addressing the five purposes introduced in Section 6 as reported in the Sankey diagram of Fig. 9. Three are for *process discovery via order relations*, they differ for the semantic engine: one is for BPMN processes, one for BPMN collaboration diagrams and the last for Petri-nets. They all use the order relation log evaluator and the trace length evaluator to avoid infinite traces (setting a threshold of 100 events per trace). Other two instantiations regard *process discovery via trace frequencies*, one simulating BPMN processes and one BPMN collaborations. They adopt the trace frequency log evaluator and the default trace evaluator since the log evaluator bounds itself to infinite traces. Similarly, PURPLE implements two instantiations of *what-if analysis via branching probabilities*, they simulate BPMN processes and collaborations, and exploit the branching log evaluator and the trace length evaluator. Finally, the last four are related to *conformance checking via noise frequencies* and *conformance checking via fixed align cost*. The first two use the noise frequency log evaluator, the others use the fixed align log evaluator, and all of them implement the trace length evaluator. All instantiations handle BPMN processes and Petri-nets.

More in detail, the three semantic engines implement BPMN process and collaboration semantics as described in [51], and the Petri-net semantics [50]. Concerning BPMN, PURPLE supports process and collaboration diagrams made up of pools, empty start, and end events, message start and end events, terminate end events, intermediate message throw and catch events, tasks, parallel

Table 3
Process discovery via order relations validation results.

Model	El.	AND	XOR	Traces	PURPLE	Coverage with 1k traces			Coverage with min traces		
						BIMP	GED	PLG2	BIMP	GED	PLG2
p0	10	1	0	3	100%	63%	100%	100%	63%	100%	100%
p1	11	1	1	3	100%	63%	100%	100%	63%	63%	75%
p2	12	1	1	5	100%	75%	100%	100%	75%	100%	100%
p3	17	1	2	5	100%	83%	100%	100%	83%	92%	100%
p4	21	1	3	10	100%	61%	100%	100%	56%	89%	100%
p5	27	1	4	10	100%	74%	91%	91%	70%	91%	83%
p6	34	3	3	14	100%	39%	69%	100%	39%	69%	94%
p7	40	6	0	76	100%	24%	68%	97%	24%	68%	93%
p8	49	6	1	226	100%	6%	49%	99%	6%	49%	97%
p9	53	4	5	41	100%	25%	54%	99%	25%	50%	89%

gateways, exclusive gateways, and event-based gateways. In particular, in the case of BPMN collaborations, PURPLE produces a unique log file, of which each trace lists the events of a single participant. For convenience, traces belonging to the same collaboration case [60] have the same case identifier. Therefore, depending on the final usage, a generated log can be used as-is, divided into separate files (one for each participant), or the traces can be merged based on the collaboration case. The latter engine, instead, supports standard Petri-nets (including particular classes of Petri-nets, such as WF-nets).

The experiments are carried out by means of synthetic and real(istic) BPMN and Petri-net models, respectively generated by PLG2,[1,2] or obtained from the literature. The models contain start/end events, activities, and XOR/AND gateways; their dimension ranges from a minimum of 8 to a maximum of 53 elements. Concerning their topology, they are both structured and unstructured, and some of them contain loops. Any further information about the models and the artifacts generated during the experiments is available at https://bitbucket.org/proslabteam/validation/.

The aim of this validation is to show the suitability of the framework in addressing mining purposes of different kinds. In each experiment, we use as a measure a quality criterion for the event logs, set on the basis of the purpose to address. When possible, we compare the results of these measurements with the ones achieved by reference tools, such as PLG2, BIMP,[3] and the ProM[4] plugin of the GED methodology [40]. We selected these tools among the ones found in the literature (we refer to Section 2 for a comprehensive review of tools for log generation) using as inclusion criteria: the availability of an operating software to be used for the experiments, and the possibility of tailoring the produced logs to the mining purpose under analysis.

7.2. Results

Following we present the result of the validation for each of the presented PURPLE instantiations.

Process discovery via order relations. For this experiment, we use *coverage*, i.e., the percentage of activity relations provided in the log with respect to the entire set of relations present in the model, as a comparison measure to assess event logs' quality. In this regard, we ran the logs generation setting to 1000, the number of traces to be produced by the tools, except for PURPLE since it stops the simulation autonomously once the purpose is satisfied. In a second experiment, for each input model, we decreased the number of traces to be produced to the number of traces that PURPLE needs to cover the entire footprint matrix. Both kinds of experiments have been repeated 10 times for each model; however, for the sake of presentation, the results reported in the following consider the worst results achieved by PURPLE and the average results achieved by the other tools. For each of the considered process models, we obtained eight event logs, two from each tool, and we compared them with respect to the coverage of the footprint matrix.

Table 3 summarizes the results of this comparison. The first two columns, *Model*, *El.*, *XOR*, and *AND* contain the name of the process model and the number of its elements, the number of XOR slpit and AND split gateways respectively. The third column, *Traces*, reports the number of traces autonomously generated by PURPLE that permit to cover the entire footprint matrix as reported in column 4. Columns 5 to 7 show the percentages of activity relations covered by BIMP, GED, and PLG2, respectively, using a threshold of 1000 traces to be generated. The last three columns provide results for analogous experiments where the values of column 3 are used as a threshold for the traces to be generated. Being guided by the evaluator, PURPLE covered entirely the relations matrix for each of the considered process models. Instead, the other tools show worse results, especially in the case of bigger models containing many parallel or exclusive branches, as such models involve higher numbers of order relations. Indeed, a model with n activities to be executed in parallel implies having $n(n-1)$ relations to discover, while a model with n activities in sequence (one after the other) shows just $n-1$ relations. For instance, model $p8$ has six parallel split gateways and one exclusive split gateway with 3 levels of nesting, and the resulting footprint matrix contains 699 relations to be discovered. The results achieved using the number of traces generated by PURPLE as threshold show that, on average, BIMP covers 6% of the footprint matrix, PLG2 97%, and GED 49%.

[1] https://plg.processmining.it/.
[2] Notably, PURPLE can take as input *.bpmn* or *.pnml* models generated by other tools, even those relying on process discovery or other techniques.
[3] https://bimp.cs.ut.ee/.
[4] https://www.promtools.org/.

Table 4
Process discovery via frequencies results.

Model	El.	AND	XOR	Traces	Loops repetition avg.	Error
p10	8	0	1	10 000	3.2	0%
p11	10	0	1	10 000	3.2	0%
p12	19	4	2	10 000	2.9	0%
p13	25	2	2	10 000	2.7	0%
p14	38	1	4	10 000	2.9	0%

Table 5
Branching probabilities validation results.

Model	El.	AND	XOR	Error with min traces			Error with 1000 traces
				Traces	BIMP	PURPLE	BIMP
p15	11	0	1	10	10%	0%	1,1%
p16	13	1	2	25	5,3%	0%	2,4%
p17	17	1	2	25	6%	0%	1,2%
p18	21	1	3	45	2%	0,4%	1,9%
p19	25	0	4	59	3,5%	0,3%	2,7%
p20	29	4	1	71	3,4%	0,7%	1,2%
p21	34	3	3	30	8,8%	0%	1,6%
p22	40	3	3	50	6%	0%	2,6%
p23	52	0	9	116	2,1%	0,5%	1,9%
p24	53	4	5	120	3,6%	0,5%	3,4%

When we increase the number of traces to produce, the results get slightly better for PLG2 which reaches 99% of coverage, while they remain unchanged for BIMP and GED.

Process discovery via frequencies. In this case, we use as quality measures the *error* in reproducing the desired percentages of occurrence for the trace variants, and the number of repetitions of each loop in the model. For this instantiation, a comparison between PURPLE and other tools would be unfair, since none of the other tools permits to customize the trace frequencies. Therefore, we run the simulations only on PURPLE. To this aim we used a set of models that contain loops, using a random value for the trace frequencies, the loop repetition thresholds fixed to 5, and the number of traces set to 10 000. We analyzed the resulting logs using ProM to extract the occurrences of each trace variant and the number of loop repetitions. The results are presented in Table 4. We report the dimension of the input model and splits gateways, the number of generated traces, and the error. For each model, PURPLE reproduces the correct number of trace variants, keeping the loop repetitions under the selected threshold. These results were expected since the evaluator always provides deltas that force the simulator to follow a precise execution trace in the LTS. Thus, the simulator produces exactly the log required by the user, avoiding errors.

What-if analysis via branching probabilities. Concerning this purpose, we take BIMP as the reference tool, because the others do not permit specifying a business scenario to simulate. We ran the log generation on a set of BPMN models from the considered dataset, using the same scenario for both tools. As a quality measure, this time we use the mean absolute *error* of the branching probabilities. In detail, we measure from the logs the percentage of appearance of each choice branch, and consequently the average of the absolute errors with respect to the scenario.

We present in Table 5 the results of the comparison. For each model, we report the number of elements and split gateways, the dimension of the log resulting from PURPLE, and the errors made by the two tools. The last column instead shows the error made by BIMP when increasing the log size to 1000 traces.

Even in this case, PURPLE better satisfies the purpose, having always an error quite close to the 0%. On average PURPLE shows a mean absolute error equal to 0,4%, while BIMP reaches 5%. It results that the logs generated in BIMP are less reliable for making assumptions on the model execution. In particular, by increasing the number of traces to 1000, see the last column of Table 5, BIMP produces smaller errors in the branching probabilities; on average, the error decreases to 2%, but it is still higher than the one of PURPLE.

Conformance checking via noise frequencies. In this case, we compare the event logs generated by PURPLE and PLG2, as the latter permits choosing percentages of noise. We compare event logs with 5000 traces and the 10% of noisy traces for each type of noise, i.e., 500 for missing head, 500 for missing tail, 500 for missing episode, 500 for order perturbation, and 500 for additional event. Finally, we analyze the logs to calculate the *error* in reproducing the desired occurrence rate for each type of noise. Table 6 reports the results of the comparison. It shows that PURPLE always produces the exact number of noised traces, while PLG2 produces fewer noised traces than requested. On average, the error in the logs of PLG2 is equal to 20,8%, meaning that around 500 noised traces over 2500 are missing. The bigger lack results in reproducing traces with order perturbation, probably because PLG2 swaps also activities that are in parallel, so that the resulting trace is still compliant with the model. This problem is avoided in PURPLE, because it checks if the noised trace is compliant or not with the model before adding it to the log.

Table 6
Conformance checking via noise frequencies results.

Model	El.	AND	XOR	Traces	Error	
					PLG2	PURPLE
p25	10	1	0	5000	20,6%	0%
p26	11	1	1	5000	22,5%	0%
p27	12	1	1	5000	21,0%	0%
p28	17	1	2	5000	21,3%	0%
p29	21	1	2	5000	18,8%	0%

Table 7
Conformance checking via fixed align cost results.

Model	Places	Transitions	Traces	Alignment cost		
				Required	Obtained	Error
p30	6	8	2000	3	3.03	1%
p31	18	19	2000	3	2.91	3%
p32	27	27	2000	3	2.93	3%
p33	35	34	2000	3	2.91	2.3%
p34	43	41	2000	3	2.89	4.3%

Table 8
PURPLE and collaboration models validation results.

Model	El.	AND	XOR	Pool	Msg.	Ord. rel.	Trace freq.	Branch. prob.
c0	33	2	1	3	4	100%	0%	0.6%
c1	20	0	2	2	5	100%	0%	0%
c2	40	1	3	4	4	100%	0%	0.9%
c3	38	0	4	3	5	100%	0%	0.7%
c4	31	3	2	3	6	100%	0%	0%

Conformance checking via fixed align cost. For this purpose, we evaluate only the logs of PURPLE, as no other tool supports this purpose. Here we set the desired alignment cost to 3 for each simulated model and a log size of 2000 traces, then we use the resulting logs and the input models to calculate, via ProM, the real costs for the alignments. Table 7 puts in comparison, for each considered model, the required and the obtained alignment costs. The results show that the generated logs have alignment costs very close to the expectations. Overall, the error percentage made by the tool is on average equal to 2.7%. This discrepancy depends on the fact that the tool generates noised traces in order to make the log converge to the required alignment cost, but before reaching it, the simulation is stopped because the requested number of traces to produce is reached.

PURPLE *and BPMN collaborations.* Since PURPLE implements a semantic engine for BPMN collaborations, and use it in three instantiations (see Fig. 9), we assessed PURPLE against five BPMN collaboration models from the literature [60]. As reported in Table 8, the models range from collaborations composed of 2 to 4 pools, 4 to 6 message flows, 20 to 41 elements, 0 to 3 AND split gateways, and 0 to 4 XOR split gateways. A complete account of the models and the logs generated from them with the three instantiations is available in the validation repository.

Concerning the results, in the rediscoverability via order relations instantiation, for all models PURPLE provides event logs that cover all order relations, including those between a sending element and its corresponding receiving element. In the rediscoverability via trace frequencies, PURPLE respects the imposed thresholds without errors. Finally, for the branching probabilities instantiation, the resulting event logs fulfill the purpose perfectly, except for a few cases where there are small errors, in line with the results in Table 5.

7.3. Performance assessment

To assess the performance of PURPLE, we measured the time the tool spent generating event logs for the introduced instantiations. We run the simulation ten times for each model and PURPLE instantiation and then report the average. As input models, we choose ten BPMN diagrams with an increasing number of elements, from 11 to 427. In particular, the last model has been selected as a borderline example since it represents an extremely large model comprising 80 parallel split gateways with 2 to 4 outgoing sequence flows. This high number of parallel gateways and parallel flows implies a large LTS. The experiments have been conducted on a MacBook Air with an M2 chip and 24 GB of RAM; precisely PURPLE was run within a Java SE 21 runtime environment, with the JVM configured to use up to 4 GB of heap space. For each instantiation, we selected a percentage of precision of 95% and for those who need them, a minimum of 1000 and a maximum of 100 000 traces. Finally, for conformance checking instantiations, we used custom noise percentages equal to 5%, and an alignment cost equal to 2.

Table 9 reports the result of the performance evaluation, each row provides model information, and the times in milliseconds needed to run the instantiation. Concerning the rediscoverability via order relations, PURPLE spent reasonable times for almost all

Table 9
Results of the performance assessment.

Model					Rediscoverability		What-if	Conformance	
Name	El.	AND	XOR	Loops	Order relations	Custom frequencies	Branching probability	Noise frequencies	Align cost
p35	11	1	1	Yes	46 ms	601 ms	62 ms	19 005 ms	6 044 ms
p36	11	2	0	No	53 ms	624 ms	32 ms	10 471 ms	3 250 ms
p37	17	2	1	Yes	88 ms	940 ms	63 ms	37 941 ms	11 793 ms
p38	27	1	4	Yes	508 ms	1 542 ms	113 ms	54 368 ms	17 238 ms
p39	34	3	3	No	382 ms	37 905 ms	1 527 ms	67 457 ms	35 674 ms
p40	39	3	3	No	1 024 ms	4 882 ms	2 585 ms	87 576 ms	43 658 ms
p42	53	4	5	No	1 481 ms	811 268 ms	8 077 ms	127 018 ms	66 167 ms
p43	111	7	7	Yes	15 413 ms	–	232 095 ms	–	693 608 ms
p44	427	80	97	Yes	3 343 174 ms	–	–	–	–

models, from a minimum of around 50 ms for a model of about ten elements, to 15 s for a model of a hundred elements (including 7 AND and 7 XOR split gateways). The time increases significantly with the last model $p44$, which consists of 427 elements (177 of which are split gateways) and loops. For this model, PURPLE took around 55 min to provide the final log. The situation changes with the next instantiations: the rediscoverability via custom frequencies requires higher times due to the extraction of the possible execution traces from the input model. Indeed, for models up to 30 elements, the tool produces the logs in about one second. For models with 30 to 50 elements, PURPLE produces the logs in reasonable times, from tens of seconds to tens of minutes, depending on the complexity of the process. For the last two models, $p43$ and $p44$, the tool suffers from traversing a very large LTS; we stopped the simulation after an hour, although there was still space in the heap memory of the JVM. For the what-if analysis via branching probabilities instantiation, PURPLE shows times of a few seconds for models up to fifty elements (8 s to process model $p42$). While for model $p43$ PURPLE took about 38 min to complete the task, and for model $p44$ it exceeded the time limit we imposed. Regarding the conformance checking via noise frequencies, the shown times are higher than all the other instantiations: from about ten/twenty seconds for models $p35$ and $p36$ to more than 2 min for model $p42$; the last two models exceeded the time limit. Finally, for conformance checking via fixed align cost, the tool terminates in reasonable times, from about 3 s for model $p35$ to 1 min for model $p42$. The outlier is model $p43$, which required about 10 min, while $p44$ exceeded the time limit.

To sum up, PURPLE achieves its purposes in reasonable times for large models up to 40 or 50 elements. Increasing the dimension to a hundred elements makes the tool unable to produce logs in a reasonable time for the instantiation with a higher computational input. Anyway, studies show that process models in BPMN notation contain on average 32 elements [61]. We are aware that our approach is less time-efficient than purpose-agnostic simulators. Indeed, the first requirement in PURPLE is to guide log generation, which requires performing heavy computations, rather than focusing on time efficiency.

8. Concluding remarks

The presented work proposes a novel framework, PURPLE, to generate event logs via guided simulation of process models. PURPLE is meant to deal with several modeling languages and different mining purposes, as well as to ensure that the produced event log brings properties related to the selected mining purpose. Along with the definition of PURPLE, we present eleven framework instantiations addressing the generation of event logs tailored to different mining purposes. These instantiations are implemented in the PURPLE tool we provide. The analysis of the related works and the comparison we conducted between the existing log generators show that PURPLE is able, better than the others, to tune the simulation to the mining purpose in reasonable times.

Assumptions and limitations. We formalize the PURPLE framework under the assumption of simple event logs, which contain only activity names. Consequently, the PURPLE framework focuses mainly on control-flow aspects. In particular, the delta inherits this assumption as it contains simple traces. This still allows for defining mining purposes and evaluators that guide the simulation according to aspects of some other model perspectives. For instance, as shown in Section 6.2, the evaluator exploits the branching probability to build up the delta and guide the simulation. In the same fashion, we could have defined an evaluator producing an event log based on the cost of the activities, e.g., an evaluator that minimizes as much as possible the average cost of the model execution. Nevertheless, handling traces with just the activity names is a limitation to the variety of mining purposes and evaluators that can be defined on top of PURPLE. For example, simple logs do not deal with the resource perspective needed for social network analysis [62], the data perspective for decision-mining purposes [63], communication for collaboration mining [60], and more in general to objects [64]. Another consequence of this assumption is that the input models cannot contain homonym activities. Indeed, without considering other information, e.g., the resource performing the activities, events triggered by different activities with the same name are impossible to distinguish thus, the guided simulation could fail when searching for states in the LTS suggested by the delta. To overcome these limitations the framework can be extended to include more sophisticated definitions of event logs and LTS transitions (c.f. Section 3) to include information about other perspectives. Moreover, even if PURPLE produces event logs containing timestamps, they correspond to the moments the tool records the events. The user cannot influence timestamps, e.g., setting activity durations and delays between activities.

Regarding the delta definition in terms of sub-traces, another concern is that it cannot guide the simulator toward more abstract or generic behaviors. For instance, the delta cannot suggest the simulator to look for traces where a loop is repeated a casual number

of times or where an event follows another not directly, as some events may appear in the middle. Instead, by defining the delta using a language for expressing a set of traces (e.g., regular expressions), we could make more complex queries on the LTS, and thus address more purposes.

Discussion. In the following, we discuss the termination guarantees of the PURPLE framework, by analyzing the PURPLE routine and the two main factors that may affect the simulation: (i) the characteristics of the input models and the related LTS, and (ii) the implementation of the log evaluator. For each factor, we illustrate the underlying challenges and the adopted solutions or assumptions.

First, PURPLE does not guarantee termination in every possible configuration. To mitigate this, we adopted mechanisms that—under specific assumptions—allow the simulation to terminate and produce a final log. In this regard, in Section 4, we have shown that PURPLE returns the final log once the main routine terminates (see purpleRoutine function in Listing 1), thus when the log evaluator return null.

Analyzing the PURPLE routine, a first aspect that influences the termination of PURPLE is related to the complexity of the input models and the consequent possibility that the PURPLE simulator is guided to perform infinite traversal of the LTS, e.g., when the model presents a loop. This weakness can be avoided via an appropriate trace evaluator that bounds the length of the trace to produce, thus avoiding infinite traversal of loops in the LTS.

Similarly, the simulator may produce an LTS of infinite dimension, e.g., in the presence of unsafe models. However, the LTS is generated on the fly during the simulation, using the getNext function (see Listing 10), without the need for the generation of the complete state space. Therefore, assuming to have an adequate trace evaluator, even if the LTS dimension can increase at every iteration of the PURPLE routine, making the computation heavily time consuming (see Section 7.3), the function globalSimulate will always terminate in a finite number of steps. Notably, this assumption implies that PURPLE produces incomplete traces or may not explore parts of the LTS required by the purpose. Anyway, the results in Section 7.2 show that this approximation does not greatly impact the completeness of the produced logs.

The second aspect that influences the termination of the PURPLE routine is the evaluation of the log. This aspect is up to the specific implementation of the evaluate function of the log evaluator instantiation. What the conceptual framework can ensure is that the input to the evaluate function, represented by the current log, is always finite. Therefore, assuming the evaluation always terminates in finite steps, the last *point of failure* is the possibility that the delta never becomes null. If this happens, the do-while loop of the PURPLE routine never ends. Also, this aspect is up to the specific implementation of the log evaluator. Anyway, the assumption that the evaluate function of the log evaluator returns null after finite iterations is not so limiting. Indeed, in the case of evaluators who continue to produce non-empty deltas due to complex requirements, it is possible to impose the termination of the PURPLE routine by bounding the number of traces to produce or lowering the strictness of the requirements. For instance, in the fixed alignment cost instantiation, when the cost of alignment for the current log tends to the required value without ever reaching it, the evaluation of the log will never produce a null delta; this explains why the instantiation requires as input also the maximum number of traces to produce.

Future work. In the future, we intend to pursue the development of the PURPLE framework, both from the theoretical and the practical point of view. We aim to formalize the PURPLE framework and its components in order to investigate its formal properties. Moreover, we intend to define and implement additional evaluators to handle other mining purposes and consider other model perspectives, like data and multi-party communication. This can, for instance, give the user the chance to generate event logs with different data quality issues to test approaches and algorithms dealing with them. Regarding the tool, we aim to parallelize the computations of the simulator by handling more than one hint of the delta at the same time. Moreover, we aim to facilitate the implementation of new semantic engines and evaluators, and thus foster collaboration with other researchers, we intend to turn the PURPLE tool into an extensible software framework, following an architecture based on plug-ins. In this way, we could divide the logic of PURPLE between a fixed core component, advocated to simulate models, and independent plug-in components, each of which implements a modeling language semantics or an evaluator. For instance, a possible plug-in can be a wrapper for integrating external formal environments, like Maude [65], to easily implement new semantic engines and increase the number of managed modeling languages. Similarly, a wrapper for process mining tools, like Disco [66] or ProM, could be developed to help the definition of new evaluators and, hence, the handling of new mining purposes.

CRediT authorship contribution statement

Andrea Burattin: Writing – review & editing, Conceptualization. **Barbara Re:** Writing – review & editing, Supervision, Funding acquisition. **Lorenzo Rossi:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Francesco Tiezzi:** Writing – review & editing, Supervision, Methodology.

Declaration of competing interest

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

Appendix. Additional listings

Function getNexts, see Listing 10, takes as input a state s and a label l. In case the state s is not in the LTS, the function returns an empty set. Otherwise, if the LTS contains s and it has not been visited yet, the semantic engine uses the function getMoves to calculate all the possible pairs of transitions outgoing from s and the relative target states. All the pairs of transitions and states are then added to the LTS, and the state s is marked as visited. Finally, the function returns the set of all states targeted by a transition outgoing from s and labeled with l if any. Only in case an outgoing transition labeled with τ exists, getNexts is called recursively on the target of the τ transition and the label l. This permits performing one or more invisible transitions before the one with the desired label.

For example, if we consider the model in Fig. 1(a) and part of its LTS in Fig. 1(d) containing only states s_i, s_1, and s_2, the function getMoves called on state s_2 and label B returns a set containing s_5. More in detail, it discovers and adds to the LTS the states s_3, s_4, s_5, s_6, s_9, and s_{10}, and their incoming transitions, but returns the only state reachable by label B.

```
1  StateSet getNexts(State s, Label l):
2      StateSet r = new StateSet();
3      if (!lts.contains(s)):
4          return r;
5      if (!lts.isVisitied(s)):
6          LTSMoves moves = getMoves(s);
7          lts.add(s, moves);
8          lts.setVisited(s);
9      for (Transition t in lts.getOutgoing(s)):
10         if(t.getLabel() == τ):
11             r.add(getNexts(t.getTarget(),l));
12         if(t.getLabel() == l):
13             r.add(t.getTarget());
14     return r;
```

Listing 10: getNext function of the Semantic Engine component.

```
1  boolean evaluate(Trace t):
2      return false;
```

Listing 11: evaluate function of the DefaultTraceEvaluator class.

```
1  int threshold;
2
3  TraceLengthEvaluator(int n):
4      threshold = n;
5
6  boolean evaluate(Trace t):
7      return t.length()> threshold;
```

Listing 12: TraceLengthEvaluator class.

```
1  float threshold;
2
3  TraceCostEvaluator(float f):
4      threshold = f;
5
6  boolean evaluate(Trace t):
7      float cost = 0;
8      for (Event e in t):
9          cost += e.getAttribute('cost');
10     return cost > threshold;
```

Listing 13: TraceCostEvaluator class.

```
1  TraceLengthEvaluator lEval;
2  TraceCostEvaluator cEval;
3
4  CustomTraceEvaluator(int n, float f):
5      TraceLengthEvaluator lEval = new TraceLengthEvaluator(n);
6      TraceCostEvaluator cEval = new TraceCostEvaluator(f);
7
8  boolean evaluate(Trace t):
9      return lEval.evaluate(t) || cEval.evaluate(t);
```

Listing 14: CustomTraceEvaluator class.

Data availability

No data was used for the research described in the article.

References

[1] W. van der Aalst, Process Mining: Data Science in Action, Springer, 2016.
[2] H. Yang, M. Park, M. Cho, M. Song, S. Kim, A system architecture for manufacturing process analysis based on big data and process mining techniques, in: IEEE International Conference on Big Data, 2014, pp. 1024–1029.
[3] A. de Medeiros, C. Günther, Process mining: Using CPN tools to create test logs for mining algorithms, in: Practical Use of Coloured Petri Nets and CPN Tools, Vol. 576, University of Aarhus, 2005, pp. 177–190.
[4] K. Cios, W. Pedrycz, R. Swiniarski, L.A. Kurgan, Data Mining: A Knowledge Discovery Approach, Springer, 2007.
[5] B. Van Dongen, A. De Medeiros, L. Wen, Process mining: Overview and outlook of petri net discovery algorithms, in: Transactions on Petri Nets and Other Models of Concurrency II, in: LNCS, Vol. 5460, Springer, 2009, pp. 225–242.
[6] W. van der Aalst, T. Weijters, L. Maruster, Workflow mining: Discovering process models from event logs, Knowl. Data Eng. 16 (9) (2004) 1128–1142.
[7] R. Bose, R. Mans, W. van der Aalst, Wanna improve process mining results? in: Computational Intelligence and Data Mining, IEEE, 2013, pp. 127–134.
[8] T. Stocker, R. Accorsi, SecSy: Security-aware synthesis of process event logs, in: Enterprise Modelling and Information Systems Architectures, 2013, pp. 71–84.
[9] R. Andrews, C. van Dun, M. Wynn, W. Kratsch, M. Röglinger, A. ter Hofstede, Quality-informed semi-automated event log generation for process mining, Decis. Support Syst. 132 (2020) 113265.
[10] A. Burattin, PLG2: Multiperspective process randomization with online and offline simulations, in: BPM Demo Track, Vol. 1789, CEUR-WS.org, 2016, pp. 1–6.
[11] E. Esgin, P. Karagoz, Process profiling based synthetic event log generation, in: IC3K, Vol. 1, SCITEPRESS, 2019, pp. 516–524.
[12] V. Kataeva, A. Kalenkova, Applying graph grammars for the generation of process models and their logs, in: Young Researchers' Colloquium on Software Engineering, Vol. 8, HSE University, 2014, pp. 83–87.
[13] A. Mitsyuk, I.S. Shugurov, A. Kalenkova, W. van der Aalst, Generating event logs for high-level process models, Simul. Model. Pr. Theory 74 (2017) 1–16.
[14] A. Burattin, B. Re, L. Rossi, F. Tiezzi, A Purpose-Guided log generation framework, in: Business Process Management, in: LNCS, Vol. 13420, Springer, Switzerland, 2022, pp. 181–198.
[15] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman, Systematic literature reviews in software engineering – A systematic literature review, Inf. Softw. Technol. 51 (1) (2009) 7–15.
[16] P. Brereton, B.A. Kitchenham, D. Budgen, M. Turner, M. Khalil, Lessons from applying the systematic literature review process within the software engineering domain, J. Syst. Softw. 80 (4) (2007) 571–583.
[17] A. Cavacini, What is the best database for computer science journal articles? Scientometrics 102 (3) (2015) 2059–2071.
[18] J. Zhu, W. Liu, A tale of two databases: the use of web of science and scopus in academic papers, Scientometrics 123 (1) (2020) 321–335.
[19] J. Webster, R.T. Watson, Analyzing the past to prepare for the future: Writing a literature review, MIS Q. 26 (2) (2002) xiii–xxiii.
[20] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: International Conference on Evaluation and Assessment in Software Engineering, in: EASE, ACM, 2014.
[21] A. Burattin, B. Re, L. Rossi, F. Tiezzi, PURPLE: a purpose-guided log generator, in: BPM Demo Track, in: LNBIP, Vol. 3299, Springer, 2022, pp. 90–94.
[22] O. Kamal, S. Sohail, F. Bukhsh, Optimizing privacy-utility trade-off in healthcare processes: Simulation, anonymization, and evaluation (using process mining) of event logs, in: Proceedings of the 14th International Conference on Simulation and Modeling Methodologies, Technologies and Applications, Scite Press, 2024, pp. 289–296.
[23] L. Garcıa-Banuelos, M. Dumas, Towards an open and extensible business process simulation engine, in: Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, 2009, pp. 199–208.
[24] E. Verbeek, D. Fahland, Generating event logs with CPN IDE, in: ICPM Doctoral Consortium and Demo Track, Vol. 3648, CEUR.ws, 2023.
[25] M. Kuhn, J. Grüger, C. Matheja, A. Rivkin, LogPPL: A tool for probabilistic process mining, in: ICPM Doctoral Consortium and Tool Demonstration Track, vol. 3783, CEUR.ws, 2024.
[26] J. Grüger, T. Geyer, D. Jilg, R. Bergmann, SAMPLE: A semantic approach for Multi-perspective event log generation, in: M. Montali, A. Senderovich, M. Weidlich (Eds.), Process Mining Workshops, Springer, 2023, pp. 328–340.
[27] D. Jilg, J. Grüger, T. Geyer, R. Bergmann, DALG: The data aware event log generator, in: Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Forum At BPM, in: CEUR Workshop Proceedings, Vol. 3469, CEUR-WS.org, 2023, pp. 142–146.
[28] L. Ackermann, S. Schönig, MuDePS: Multi-perspective declarative process simulation, in: BPM Demo Track, Vol. 1789, CEUR.ws, 2016, pp. 23–27.
[29] V. Skydanienko, C. Di Francescomarino, C. Ghidini, F. Maggi, A tool for generating event logs from Multi-Perspective declare models, in: BPM Demo Track, Vol. 2196, CEUR.ws, 2018, pp. 111–115.
[30] C. Di Ciccio, M.L. Bernardi, M. Cimitile, F.M. Maggi, Generating event logs through the simulation of declare models, in: Enterprise and Organizational Modeling and Simulation, in: LNBIP, Vol. 231, Springer, 2015, pp. 20–36.
[31] A. Alman, C. Di Ciccio, D. Haas, F. Maggi, J. Mendling, Rule mining in action: The RuM toolkit, in: ICPM Doctoral Consortium and Tool Demonstration Track, Vol. 1789, CEUR.ws, 2016, pp. 51–54.
[32] A. Alman, F.M. Maggi, M. Montali, A. Rivkin, Generating event logs from hybrid process models, in: Business Process Management Workshops, Springer, 2024, pp. 289–301.
[33] I. Donadello, F.M. Maggi, F. Riva, M. Singh, ASP-Based log generation with purposes in Declare4Py, in: ICPM Doctoral Consortium and Demo Track, Vol. 3648, CEUR.ws, 2023.
[34] D. Sommers, N. Sidorova, B. van Dongen, A ground truth approach for assessing process mining techniques, Process. Sci. 2 (1) (2025).
[35] R. Accorsi, T. Stocker, SecSy: Synthesizing process event logs, in: Enterprise Modelling and Information Systems Architectures, Gesellschaft für Informatik e.V., 2013, pp. 71–84.
[36] T. Stocker, R. Accorsi, SecSy: A security-oriented tool for synthesizing process event logs, in: BPM Demo Track, Vol. 1295, CEUR.ws, 2014, p. 71.
[37] M. Riess, SynBPS: a parametric simulation framework for the generation of event-log data, Simulation 100 (8) (2024) 849–870.
[38] F. Meneghello, C. Di Francescomarino, C. Ghidini, Runtime integration of machine learning and simulation for business processes, in: International Conference on Process Mining, IEEE, pp. 23–27.
[39] A. Maldonado, C.M.M. Frey, G.M. Tavares, N. Rehwald, T. Seidl, GEDI: Generating event data with intentional features for benchmarking process mining, in: Business Process Management, in: LNCS, Vol. 14940, Springer, 2024, pp. 221–237.
[40] T. Jouck, B. Depaire, Generating artificial data for empirical analysis of control-flow discovery algorithms: A process tree and log generator, Bus. Inf. Syst. Eng. 61 (6) (2019) 695–712.
[41] T. Jouck, B. Depaire, PTandLogGenerator: a generator for artificial event data, in: BPM Demo Track, Vol. 1789, CEUR.ws, 2016.

[42] J. Grimm, A. Kraus, H. van der Aa, CDLG: A tool for the generation of event logs with concept drifts, in: BPM Demo Track, vol. 3216, CEUR.ws, 2022, pp. 92–96.
[43] D. Loreti, F. Chesani, A. Ciampolini, P. Mello, Generating synthetic positive and negative business process traces through abduction, Knowl. Inf. Syst. 62 (2) (2019) 813–839.
[44] W. van der Aalst, Matching observed behavior and modeled behavior: An approach based on Petri nets and integer programming, Decis. Support Syst. 42 (3) (2006) 1843–1859.
[45] OMG, Business process model and notation (BPMN V 2.0), 2011.
[46] M. Dumas, M. La Rosa, J. Mendling, H. Reijers, Fundamentals of Business Process Management, Springer, 2013.
[47] W. van der Aalst, Process mining in the large: A tutorial, in: Business Intelligence, in: LNBIP, Vol. 172, Springer, ISBN: 9783319054605, 2014, pp. 33–76.
[48] F. Corradini, C. Muzi, B. Re, L. Rossi, F. Tiezzi, MIDA: Multiple instances and data animator, in: BPM Demo, Vol. 2196, CEUR-WS.org, (ISSN: 16130073) 2018, pp. 86–90.
[49] B. Abdul, F. Corradini, B. Re, L. Rossi, F. Tiezzi, UBBA: Unity based BPMN animator, in: CAiSE Forum, in: LNCS, Vol. 350, Springer, 2019, pp. 1–9.
[50] J. Meseguert, U. Montanari, V. Sassonet, On the semantics of petri nets, in: Conference on Concurrency Theory, in: LNCS, Vol. 630, Springer, 1992, pp. 286–301.
[51] F. Corradini, C. Muzi, B. Re, L. Rossi, F. Tiezzi, Formalising and animating multiple instances in BPMN collaborations, Inf. Syst. 103 (2022).
[52] F. Corradini, J. Piccioni, B. Re, L. Rossi, F. Tiezzi, On the interplay between BPMN collaborations and the physical environment, in: Business Process Management, in: LNCS, Vol. 14940, Springer, 2024, pp. 93–110.
[53] G. De Giacomo, M. Dumas, F.M. Maggi, M. Montali, Declarative process modeling in BPMN, in: Advanced Information Systems Engineering, in: LNCS, Vol. 9097, Springer, 2015, pp. 84–100.
[54] R.M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inf. Softw. Technol. 50 (12) (2008) 1281–1294.
[55] A. Weijters, W. van Der Aalst, A. De Medeiros, Process Mining with the Heuristics Miner-Algorithm, TU/E, Tech. Rep., 166, 2006, pp. 1–34.
[56] J.L. Pereira, A.P. Freitas, Simulation of BPMN process models: Current BPM tools capabilities, in: New Advances in Information Systems and Technologies, in: AISC, Vol. 444, Springer, 2016, pp. 557–566.
[57] A. Adriansyah, Aligning Observed and Modeled Behavior (Ph.D. thesis), Mathematics and Computer Science, ISBN: 978-90-386-3574-3, 2014, http://dx.doi.org/10.6100/IR770080.
[58] C. Günther, Process Mining in Flexible Environments (Ph.D. thesis), in: Beta dissertations, Technische Universiteit Eindhoven - Industrial Engineering and Innovation Sciences, 2009.
[59] V. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, in: Soviet Physics Doklady, Vol. 10, 1966, pp. 707–710.
[60] F. Corradini, S. Pettinari, B. Re, L. Rossi, F. Tiezzi, A technique for discovering BPMN collaboration diagrams, Softw. Syst. Model. (2024) 1–21.
[61] I. Compagnucci, F. Corradini, F. Fornari, B. Re, A study on the usage of the BPMN notation for designing process collaboration, choreography, and conversation models, Bus. Inf. Syst. Eng. 66 (1) (2023) 43–66.
[62] W.M.P. van der Aalst, M. Song, Mining social networks: Uncovering interaction patterns in business processes, in: Business Process Management, in: LNCS, Vol. 3080, Springer, Germany, 2004, pp. 244–260.
[63] A. Rozinat, W.M.P. van der Aalst, Decision mining in ProM, in: Business Process Management, in: LNCS, Vol. 4102, Springer, Germany, 2006, pp. 420–425.
[64] W. van der Aalst, Object-Centric process mining: Unraveling the fabric of real processes, Mathematics 11 (12) (2023).
[65] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, The MAUDE 2.0 system, in: International Conference on Rewriting Techniques and Applications, in: LNCS, Vol. 2706, Springer, 2003, pp. 76–87.
[66] C. Günther, A. Rozinat, DISCO: Discover your processes, in: BPM Demo Track, in: CEUR Workshop Proceedings, Vol. 940, 2012, pp. 40–44.

Andrea Burattin has been an Associate Professor at the Technical University of Denmark since April 2019. Previously, he worked as an Assistant Professor at the same university, and as a postdoctoral researcher at the University of Innsbruck (Austria) and at the University of Padua (Italy). In 2013 he obtained his Ph.D. degree from a joint Ph.D. School between the University of Bologna and Padua (Italy). His Ph.D. thesis received the Best Process Mining Dissertation Award from the IEEE Task Force on Process Mining. He is a member of the steering committee of the IEEE Task Force on Process Mining.

Barbara Re is an Associate Professor of Computer Science at the University of Camerino. She received her Ph.D. in Information Science and Complex Systems from the University of Camerino. Her main research interests are in Business Process Management and Process Mining with a focus on collaborative processes. Particular attention is paid to pushing the use of formal methods as methodological and automatic tools for the development of high-quality process-aware information systems. She was involved in multidisciplinary research projects in collaboration with national and international research institutes and companies.

Lorenzo Rossi is a Post-doc at the University of Camerino (Italy), where he earned a Ph.D. in Computer Science in 2019. He also won different scholarships, at the same institution, for conducting research on Business Process Animation. His main research interests are in Business Process Management and Process Mining with a focus on collaborative processes. He aims to provide software solutions, with a solid base on theoretical computer science, for managing business processes.

Francesco Tiezzi is Associate Professor of Computer Science at the Università degli Studi di Firenze. He received the Laurea degree from the Università degli Studi di Firenze and the Ph.D. degree from the same university. His research activities focus on modeling and programming languages, the foundational study of distributed systems, and software engineering methodologies and tools based on formal methods for developing, analyzing, and managing business processes.