A Purpose-Guided Log Generation Framework

Andrea Burattin[1], Barbara Re[2], Lorenzo Rossi[2,†], and Francesco Tiezzi[3]

[1] Technical University of Denmark, Kgs. Lyngby, Denmark
andbur@dtu.dk
[2] School of Science and Technology, University of Camerino, Italy
{barbara.re,lorenzo.rossi}@unicam.it
[3] Dipartimento di Statistica, Informatica, Applicazioni, University of Florence, Italy
francesco.tiezzi@unifi.it

Abstract. Process mining is a prominent discipline that collects a variety of techniques fulfilling different mining purposes by gathering information from event logs. This involves the continuous necessity of event logs suitable for testing mining techniques with respect to different purposes. Unfortunately, event logs are hard to find and usually contain noise that can influence the results of a mining technique. In this paper, we propose a framework for generating event logs tailored for different mining purposes, e.g., process discovery and conformance checking. Event logs generation and tuning are made out through business model simulations guided by the mining purpose under consideration. Beyond defining the framework, we implemented it as a tool, which has been also used for the validation of the approach we propose.

Keywords: Process mining · Event log · Log generation · Simulation

1 Introduction

Nowadays, process mining is recognized as 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 [2]. The importance of process mining is well recognized also by companies, which appreciate the possibility to gather knowledge from their processes from actual execution data [24].

Process mining is a family of techniques and algorithms that enables to automatically extract information out of 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 testing them against different event logs [17], usually coupled with the models that generated them [9]. Mining algorithms extract different types of information according to the mining purpose they have to accomplish, e.g., process discovery and conformance checking. Therefore, to test a process mining algorithm it is important to use event logs that suit the purpose for

† Main contributor and corresponding author.

which the algorithm has been devised [22]. 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 [3]), then a fair comparison of the algorithms would require logs where such properties are indeed satisfied. As stated in the literature [7,17,21], each of those purposes heavily relies on the quality, with respect to specific properties, of the event logs given as input to the related mining algorithms.

Obtaining event logs fitting for a purpose is a complex, yet necessary, achievement [6]. Specifically, [7] claims that 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. Event logs are difficult to find, in particular those directly extracted from deployed IT systems that refer to real-world installations [8]. In this regard, several approaches, e.g., [8,12,15,17,19], 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 every time a different event log. The above-mentioned issue paves the way to the need of answering the following research questions:

RQ1: *Is it possible to define an approach for the automated generation of event logs tailored to different mining purposes?*

RQ2: *Can model simulation be guided to produce event logs that fulfill a mining purpose better than the ones generated with purpose-agnostic simulations?*

To address these research questions, we propose the PURpose-Guided Log gEneration (PURPLE) framework. The main advantages of the PURPLE framework with respect to existing simulators are as follows. PURPLE generates event logs specifically tailored to the purpose of the mining technique under investigation. To shape out 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 on the basis of the partial log generated up to that moment and the properties required by the mining purpose. Additionally, the framework is meant to *simulate many kinds of business process model*s (e.g., BPMN, Petri Net, WF-net). Besides the framework, we provide the PURPLE tool, which implements a BPMN and a Petri-net semantic engine, and addresses mining purposes concerning process discovery and conformance checking. To 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 the ones of other log generators.

The rest of the paper is structured as follows. Section 2 provides notions on event logs and Labeled Transition Systems. Section 3 introduces the PURPLE framework. Section 4 presents the PURPLE tool and several instantiations of the framework, while Section 5 reports the results of the conducted experiments. Section 6 compares our approach with related works. Finally, Section 7 closes the paper discussing assumptions, limitations, and opportunities for future works.

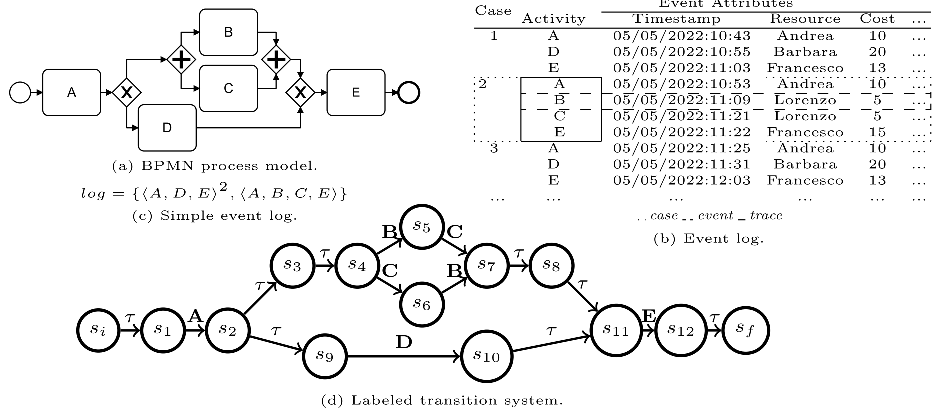
2 Background Notions

This section provides notions we use in the rest of the paper. An **event log** consists of a set of **cases**, each of which refers to some events that can be seen as one possible run of the process. An **event** refers to the execution of a system activity, and it is described by a set of **attributes**. The most common attributes for a recorded event are the *activity name* and the *timestamp*, 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**.

Figure 1(a) depicts a system modeled using the BPMN notation [20], Figure 1(b) shows a table containing an event log fragment with three cases generated by the BPMN model, while Figure 1(c) reports a *simple event log* [2, Ch. 5], which focuses only on the names of the executed activities. In this situation, an event log can be thought of as a multiset of traces, where a *trace* is a sequence of activity names [2]. 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 [8]. 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 [10,4], to execute models [2, 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 [18] or the transition rules of BPMN operational semantics [11]) usually defined by means of Labeled Transition Systems (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. 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)* $(s_1, l_1, s_2) \in\rightarrow$, $(s_2, l_2, s_3) \in\rightarrow$, ..., $(s_m, l_m, s_{m+1}) \in\rightarrow$ 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**. Fig-



(a) BPMN process model.

Case	Activity	Timestamp	Resource	Cost	...
		Event Attributes			
1	A	05/05/2022:10:43	Andrea	10	...
	D	05/05/2022:10:55	Barbara	20	...
	E	05/05/2022:11:03	Francesco	13	...
2	A	05/05/2022:10:53	Andrea	10	...
	B	05/05/2022:11:09	Lorenzo	5	...
	C	05/05/2022:11:21	Lorenzo	5	...
	E	05/05/2022:11:22	Francesco	15	...
3	A	05/05/2022:11:25	Andrea	10	...
	D	05/05/2022:11:31	Barbara	20	...
	E	05/05/2022:12:03	Francesco	13	...
...	...	...	...	...	...

(b) Event log.

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

(c) Simple event log.

(d) Labeled transition system.

Fig. 1. From process model to event log.

ure 1(d) reports the LTS representing the behavior of the model in Figure 1(a) produced by the BPMN formalization described in [11]. Each configuration of the 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 model is rendered in the LTS through a transition labeled by the name of the activity. Traversing the LTS from state s_i to s_f, the sequences of visible labels associated to the transitions represent the execution traces that can be generated from the BPMN model in Figure 1(a).

3 PURPose-guided Log gEneration Framework

In this section, we introduce the PURPose-Guided Log gEneration (PURPLE) framework. It is meant to produce, by simulating models, event logs with different properties for targeting different mining purposes. PURPLE supports the simulation of models specified with different languages, by projecting their execution onto a common behavioral model, i.e., an LTS. Figure 2 depicts the components of the PURPLE framework: a **semantic engine**, an **evaluator**, and a guided **simulator**. Except for the simulator that is fixed, the other components can be instantiated with different semantic engines, each one supporting a given modeling language (e.g., BPMN, Petri Net), and with different evaluators, each one tailored to a mining purpose (e.g., process discovery, compliance checking).

Before presenting in detail the PURPLE components, we introduce here the concept of *context* in which the PURPLE components act. The context collects and keeps updated the (even partial) LTS and a log of the model under consideration. It acts as a sort of global variable that the PURPLE components can access/modify during the simulation like in a side-effect function. Notably, at the beginning of a simulation, the context is set to an initial configuration where the log is empty and an LTS contains only the initial state s_i.

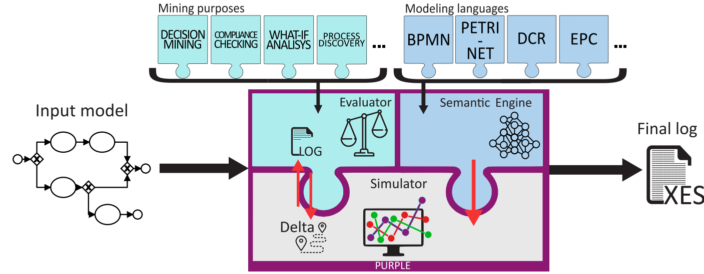


Fig. 2. PURPLE framework components.

We can now define the PURPLE components starting from the **semantic engine**. Being aware of its formal semantics, this component enacts the input model. Given a state of the corresponding LTS (i.e., a model configuration), the semantic engine returns the next reachable states (i.e., the model configurations reachable by one move), and the labels of the transitions leading to them (i.e., the names of the performed activities). For example, considering the LTS in Figure 1(d) and its state s_4, the semantic engine returns the next reachable states coupled with the labels of the connecting transitions: $\{(B, s_5), (C, s_6)\}$. Relying on different semantic engines, PURPLE can obtain from each business process model with an executable semantics the corresponding LTS [2, Ch. 3], thus gaining in terms of generalizability.

The second component is the **simulator** that is devoted to produce traces from the execution of the input model. By invoking the semantic engine, the simulator component incrementally adds states and transitions to an LTS, then traverses it to produce new traces to be added in the log. Notably, the simulator acts on the LTS and on the log maintained by the context, starting from the initial context. While building the LTS, the simulator generates a partial view of the entire LTS and, at any time it reaches a final state, it stores the visited trace in the log. The peculiarity of this component lies in a guided traversal of the LTS to guarantee the production of traces, and hence a log, that satisfy the desired mining purpose. Indeed, differently from a purely-random simulation, what the framework proposes is a **guided simulation** that takes as input a *guide* suggesting an execution path, or part of it, to follow in the LTS traversal.

Lastly, the **evaluator** component is responsible to evaluate an event log in relation to the peculiarities of the desired mining purpose. More practically, by checking 'how much' the event log stored in the context satisfies the properties needed for the purpose under consideration, the evaluator produces a *delta*, i.e., the guide that drives the simulator. A delta consists of sub-traces of the LTS that have to be added in the log to increase its suitability for the purpose. These sub-traces act as a bias for the simulation indicating the parts of the LTS to be traversed, thus influencing the produced traces. As each evaluator is defined to deal with a specific mining purpose, the generated delta is defined to achieve in the final event log the properties required by that purpose. We clarify this point with a simplistic example, used just for the sake of presentation: considering a mining purpose that requires a log in which model activities appear at least once,

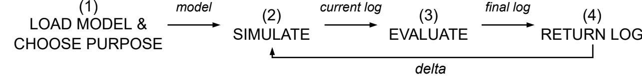


Fig. 3. PURPLE routine.

the 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.

By fixing a modeling language and a mining purpose (hence, a semantic engine and an evaluator, respectively), we get an instantiation of PURPLE ready for producing logs. Providing a model as input, the PURPLE instantiation starts performing the looping four steps routine depicted in Figure 3. Step (1) loads the input model and sets the initial context. Then, the framework's routine loops between Steps (2) and (3) before producing the final log in Step (4).

```
1  SIMULATE(st)  :
2      if  st = ⟨⟩
3          return  RANDOMSIM()
4      States := FIND(lts, st[1])
5      st := st \ st[1]
6      for  s  in  States
7          st_p := GETPREFIX(lts, s)
8          t := GUIDEDSIM(st_p, s, st)
9          if  t ≠ ⟨⟩
10             return  t
11     return  ⟨⟩
```

Listing 1.1. Simulation function for a given context $\langle lts, log\rangle$.

```
1  GUIDEDSIM(st_p, s_curr, st)  :
2    if  st = ⟨⟩
3        return  FINALISE(st_p)
4    States := FIND(lts, st[1])
5    for  s_next  in  States
6      if  ∃ (s_curr, st[1], s_next) ∈ →
7          st_p := st_p + st[1]
8          t := GUIDEDSIM(st_p, s_next, st\st[1])
9          if  t ≠ ⟨⟩
10             return  t
11   return  ⟨⟩
```

Listing 1.2. Giuded simulation for a given context $\langle lts, log\rangle$.

Step (2) performs the guided simulation of the model taking into consideration the context containing the current LTS and log. As shown in Listing 1.1, the simulation function depends on the input parameter st that is one of the sub-traces in the delta. In case st is empty (i.e., $\langle\rangle$), the function executes the model in a random way (line 3) via the RANDOMSIM() call: starting from the initial state of the LTS contained in the context, it repetitively invokes the semantic engine to know the next states (adding them to the LTS in the context) and chooses one of them randomly until it reaches one of the final states. This happens, for instance, in the case the evaluator has not performed any comparison yet. In case of non-empty delta, instead, the function proceeds by considering st as breadcrumbs to follow for logging a specific trace in the LTS. More in detail, function FIND($lts, st[1]$) (line 4) returns a set of states of the LTS reachable by a transition labeled as the first element in the considered sub-trace, i.e., $st[1]$. In case the found states do not have any successor in the current LTS, the function FIND invokes the semantic engine and adds the results in the LTS. For each found state s (line 6), the simulator calculates a prefix sub-trace st_p that leads to s via function GETPREFIX(lts, s) (line 7). Then, the algorithm calls the recursive function GIUDEDSIM (line 8) to complete the trace with labels corresponding to the remaining part of st, where the first label has been removed (line 5). The guided simulation function, Listing 1.2, takes as input the prefix sub-trace st_p, the current state s_{curr} of the LTS, and the remaining part of the hint of the delta, i.e., the sub-trace st. This function recursively searches for states of the

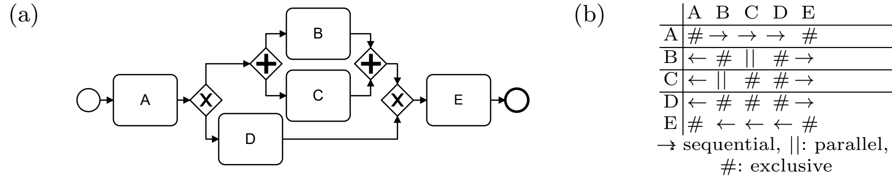
LTS in the context, reachable from s_{curr} through a sequence of transitions labeled by the remaining elements in the hint st (lines 5-10). If a reachable state is found (line 6, where $\rightarrow$ is the transition relation of the LTS), the prefix trace is increased with the label of the connecting transition (line 7, where + denotes the append operator on sub-traces). Then, the function is called recursively on the enriched prefix, the next configuration, and the hint without the first label (line 8). Once st no longer contains labels (line 2), the function enacts the base case (line 3) where function FINALISE(st_p) finalizes the prefix trace logging the labels of the transitions leading to a final state, and returns the entire trace to the calling function.

In Step (3) of the routine, the evaluator uses the context containing the event log produced by the simulator in Step (2). On the basis of the mining purpose, a specific evaluator calculates the delta, and evaluates if the purpose is satisfied. If not, the routine loops back to Step (2) to repeat a new simulation based on the calculated delta. Instead, if the purpose is satisfied, the simulation terminates and the generated event log is given as output in Step (4).

4 PURPLE at work

We present here four instantiations of the PURPLE framework addressing purposes concerning process discovery and conformance checking. These instantiations are described using the PURPLE tool that implements the framework and its routine. Tool, source code, instructions, and examples are available at `https://pros.unicam.it/purple/`. The PURPLE tool provides two semantic engines that implement a wide subset of the BPMN semantics described in [11], and the Petri-net semantics [18]. Concerning BPMN, PURPLE supports process and collaboration diagrams made up by pools, empty start and end events, message start and end events, terminate end events, intermediate message throw and catch events, tasks, parallel gateways, exclusive gateways, and event-based gateways. The latter engine, instead, supports standard Petri-nets (including particular classes of Petri-nets, such as WF-nets). Moreover, PURPLE implements four evaluators addressing process discovery and conformance checking. Notice that some evaluators may require, besides the log, additional parameters dealing with specific implementation aspects (e.g., a maximum number of traces to generate for ensuring termination). The pseudocode of the four evaluators is available online, at the PURPLE's website, in a companion technical report.

***Process discovery in* PURPLE.** The first instantiation of PURPLE that we consider regards the process discovery. To check the reliability of a discovery algorithm, or to perform a benchmark of different techniques, logs presenting specific characteristics are required. PURPLE implements evaluators addressing two specific discovery purposes: one is devised for algorithms relying on the order relation between activities, such as the Alpha algorithm [3], while the other one is for algorithms relying on frequencies, such as the Heuristics miner [23]. All these purposes can be applied to both BPMN and Petri-net models. In the rest of the section, for the sake of presentation, we consider only BPMN models,



	A	B	C	D	E
A	#	→	→	→	#
B	←	#	$\parallel$	#	→
C	←	$\parallel$	#	#	→
D	←	#	#	#	→
E	#	←	←	←	#

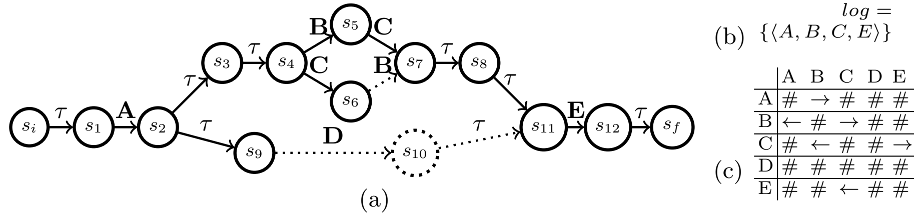
→ sequential, $\parallel$: parallel, #: exclusive

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

but the same reasoning applies to Petri-net ones, with the only difference in the generation of the LTS by means of a different semantic engine.

The aim of the **Process discovery via order relations** purpose is to 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, scans 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 [3, 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 $X||Y$, means that X directly follows Y and vice versa ($X||Y \iff X > Y$ 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 Figure 4(a), the corresponding matrix is provided in Figure 4(b). 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 with PURPLE through an evaluator that guides the simulation into the discovery of the footprint matrix avoiding to produce duplicates of the same trace. Therefore, PURPLE points at generating the smallest log covering the relations in the footprint matrix.

The simulation step of the routine is triggered at first with an empty *delta*, leading to a random simulation of the model. A possible trace result of the first simulation run maybe $\langle A, B, C, E \rangle$, where the simulator performed tasks A, B, C and E, one after the other modifying the initial context. Specifically, the simulator adds to the initial LTS the states and the transitions discovered by the semantic engine, producing the LTS in Figure 5(a) to the exclusion of dotted states and transitions which are still to discover. Moreover, it inserts in the empty log the discovered trace, resulting in Figure 5(b). Notably, to speed up the generation of the entire LTS, the simulation adds to it all the states discovered by the semantic engine, even if they do not take part to the produced trace (see states s_6 and s_9). Then, the evaluator calculates the order relations considering the updated log in the context. 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 Figure 5(c). At this point, PURPLE compares the obtained footprint matrix with the one of the original model (Figure 4(b)) to calculate the missing relations, and produces the delta for the upcoming simu-



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

Fig. 5. LTS (a), log (b), and footprint (c) resulting from the first run of simulation.

lation step. The order relations 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 in the search of additional traces containing the missing relations; in doing that, the simulator relies also on the LTS in the context. 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 $\langle A, C, B, E\rangle$. The LTS produced by the simulator after the second run of simulation corresponds to the one in Figure 5(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 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. The purpose is satisfied since the log covers all relations and does not contain repeated traces, hence PURPLE produces as output the *.xes* file.

The **Process discovery via frequencies** instantiation aims at generating 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 of an order relation. To this aim, we provide an instantiation of PURPLE permitting to choose the traces frequency. The resulting event log can be tuned in order to represent more realistic situations where behaviors could be less or more frequent than others. Logs of that form suite also for comparing the filtering approaches of different algorithms. To address this purpose, PURPLE extracts the set of traces the model can perform and information regarding the loops. Then, the user specifies the 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. Therefore, during the log generation, the 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. In case some of these values are lower than requested, the evaluator passes to the simulator a delta containing the entire traces that are still infrequent in the log. If a trace contains a loop, the 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, PURPLE returns the log *.xes* file.

***Conformance checking in* PURPLE.** Lastly, we present purposes related to conformance checking, a family of techniques for comparing a model and a log. In particular, we consider techniques based on [2]. 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 are the involved activities. Computing alignments is an expensive task, especially in presence of models with huge state-space, and there exist different approaches implementing it [5]. 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.

The **conformance checking via noise frequencies** instantiation 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 [13]; 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 instantiation of the framework 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 evaluator sends an empty delta to the simulator to receive back a random trace without noise. Then, it 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 case of *order perturbation* it swaps two or more events in the trace, while in case of *additional event* it inserts an event named differently from every activity name in the model. Once the evaluator finds the desired noise percentages and number of traces, it returns the final log.

The **conformance checking via fixed align cost** purpose 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 Figure 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, PURPLE 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, the framework extracts from the model the set of traces that can be produced, and uses them later for calculating the alignment costs. Then, similarly to the previous purpose, the 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 evaluator calculates the optimal alignment cost computing the minimum among the Levenshtein distances [16] between the noisy trace and traces previously extracted from the model.

5 Validation

In this section, we present a list of experiments on the presented instantiations of the framework, using the corresponding implementations in the PURPLE tool. The experiments are carried out by means of synthetic and real(istic) BPMN and Petri-net models, respectively generated by PLG2 (`https://plg.processmining.it/`) 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/`. Notably, 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 (`https://bimp.cs.ut.ee/`), and the ProM (`https://www.promtools.org/`) plugin of the GED methodology [14]. We selected these tools among the ones found in the literature (we refer to Section 6 for a comprehensive review of tools for log generation) using as inclusion criteria: the availability of an operating

Model	El.	Traces	PURPLE	Coverage with 1000 traces			Coverage with min traces		
				BIMP	GED	PLG2	BIMP	GED	PLG2
p0	10	3	100%	63%	100%	100%	63%	100%	100%
p1	11	3	100%	63%	100%	100%	63%	63%	75%
p2	12	5	100%	75%	100%	100%	75%	100%	100%
p3	17	5	100%	83%	100%	100%	83%	92%	100%
p4	21	10	100%	61%	100%	100%	56%	89%	100%
p5	27	10	100%	74%	91%	91%	70%	91%	83%
p6	34	14	100%	39%	69%	100%	39%	69%	94%
p7	40	76	100%	24%	68%	97%	24%	68%	93%
p8	49	226	100%	6%	49%	99%	6%	49%	97%
p9	53	41	100%	25%	54%	99%	25%	50%	89%

Table 1. Process discovery via order relations validation results.

software to be used for the experiments, and the possibility of tailoring the produced logs to the mining purpose under analysis.

Regarding the **process discovery via order relations,** the comparison measure we use to assess event logs quality is *coverage*, i.e., the percentage of activity relations provided in the log with respect to the entire set of relations present in the model. In this regard, we ran the logs generation setting to 1000 the number of traces to produce by the tools, except for PURPLE since it stops autonomously the simulation once the purpose is satisfied. In a second experiment, for each input model we decreased the number of traces to be produced to the amount of traces that PURPLE needs to cover the entire footprint matrix. Notably, both kinds of experiments have been repeated 10 times for each model, but, 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 model, we obtained eight event logs, two from each tool, and we compared them with respect to the coverage of the footprint matrix. Table 1 summarizes the results of this comparison. The first two columns, *Model* and *El.*, contain the name of the process model and the number of its elements, 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 from 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 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 the 6% of the footprint matrix, PLG2 the 97%, and GED the 49%. When we increase the number of

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

For what concerns the **process discovery via frequencies**, 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 10000. 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 2. We report the dimension of the input model, 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.

Model	El.	Traces	Loops repetition avg.	Error
p10	8	10000	3.2	0%
p11	10	10000	3.2	0%
p12	19	10000	2.9	0%
p13	25	10000	2.7	0%
p14	38	10000	2.9	0%

Table 2. Process discovery via frequencies results.

For the **conformance checking via noise frequencies**, we compare the event logs generated by PURPLE and PLG2, as the latter permits to choose percentages of noise. We compare event logs with 5000 traces and the 10% of noised 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 3 reports the results of the comparison. It shows that PURPLE produces always 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.

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

Table 3. Conformance checking via noise frequencies results.

With respect to the **conformance checking via fixed align cost**, 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 4 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.

			Alignment cost		
Model	El.	Traces	Required	Obtained	Error
p30	6	2000	3	3.03	1%
p31	18	2000	3	2.91	3%
p32	27	2000	3	2.93	3%
p33	35	2000	3	2.91	2.3%
p34	43	2000	3	2.89	4.3%

Table 4. Conformance checking via fixed align cost results.

6 Related works

This section discusses the most relevant works on the generation of artificial event logs. In describing them, we put the focus on the main features of PURPLE, such as the generation of event logs tailored to a desired mining purpose from models specified in different modeling languages. Thus, we mainly consider which kinds of event logs these approaches can generate, and which models they support.

Esgin and Karagoz present in [12] a solution to the problem of unlabeled event logs [1] 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 performs random executions of the input Petri-net. Differently from us, the approach supports only Petri-nets, cannot handle different mining purposes, and is not implemented in a tool.

Kataeva and Kalenkova propose in [15] grammar rules generating well-structured WF-Nets from which to produce logs. With respect to PURPLE, 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. In the same fashion, Burattin presents in [8] 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. With respect to PURPLE, this approach relies on random executions of the input model and works only with BPMN. Similarly, Alves and Günter propose in [17] a tool for the generation of event logs through the simulation of colored Petri-nets. They point out the issues related to the use of 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. Also this approach cannot tune the logs to produce since it relies on a random simulation of the input model.

Mitsyuk et al. face in [19] 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. With respect to our work, they can simulate communication between processes; however their approach only deals with a single modeling language and cannot be tuned for specific purposes. Stocker and Accorsi introduce in [21] 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 in order to violate security properties. Compared to ours, this approach takes care of just one specific purpose for which the produced event logs are tuned, and of just one modeling language (i.e., Petri-net).

Finally, Jouck and Depaire present in [14] a log generation approach specific for the comparison of discovery algorithms. They produce, and then simulate, a population of well-structured models from selected workflow patterns, to ensure the presence of specific activities order relations chosen by the user. Compared to ours, this work uses process trees to produce logs and, apart from process discovery, further purposes are not taken in consideration.

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

7 Concluding Remarks

The presented work proposes a novel framework, PURPLE, to generate event logs via guided simulation of business 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 framework instantiations addressing the generation of event logs tailored to four purposes. These instantiations and two semantic engines for BPMN and Petri-Net 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 conclusion, both research questions presented in the Introduction can be positively answered. Concerning **RQ1**, we provided a general framework for the automated generation of event logs tailored to different mining purposes, as

well as several instantiations of it, thus proving the feasibility of the approach. Regarding **RQ2**, we experimented our solution considering different purposes, and it proved to be more effective compared to purpose-agnostic simulators.

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 the sub-traces included in the delta are lists of activity names. Notably, this still allows defining mining purposes and evaluators that guide the simulation according to aspects of some other model perspectives. For instance, one can define an evaluator producing log on the basis of the cost of the activities tailored to what-if analysis techniques. Nevertheless, handling traces with just the activity names results in 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 purposes, or the data perspective for decision mining purposes. 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.

Future Works. As future works, 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 other evaluators, in order to handle other mining purposes and take into account other model perspectives, like data and multi-party communication. This can, for instance, give the chance to the user to generate event logs with different data quality issues in order to test approaches and algorithms dealing with them. Regarding the tool, we aim at parallelizing the computations of the simulation by handling more than one hint of the delta at the same time, and we plan to implement a debugging console for spotting useful information on simulations, including possible tool anomalies.

Acknowledgments. Work partially supported by the Italian MIUR PRIN projects *Seduce* n. 2017TWRCNB and *Fluidware* n. 2017KRC7KT, and the INdAM GNCS 2020 project *Sistemi reversibili concorrenti: dai modelli ai linguaggi*.

References

1. van der Aalst, W.: Matching observed behavior and modeled behavior: An approach based on Petri nets and integer programming. Decision Support Systems **42**(3), 1843–1859 (2006)

2. van der Aalst, W.: Process mining: data science in action. Springer (2016)
3. van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. Knowledge and Data Engineering **16**(9), 1128–1142 (2004)
4. Abdul, B., Corradini, F., Re, B., Rossi, L., Tiezzi, F.: UBBA: Unity based BPMN animator. In: CAiSE Forum. LNCS, vol. 350, pp. 1–9. Springer (2019)
5. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. thesis, Mathematics and Computer Science (2014). https://doi.org/10.6100/IR770080
6. Andrews, R., van Dun, C., Wynn, M., Kratsch, W., Röglinger, M., ter Hofstede, A.: Quality-informed semi-automated event log generation for process mining. Decision Support Systems **132**, 113265 (2020)
7. Bose, R., Mans, R., van der Aalst, W.: Wanna improve process mining results? In: Computational Intelligence and Data Mining. pp. 127–134. IEEE (2013)
8. Burattin, A.: PLG2: Multiperspective Process Randomization with Online and Offline Simulations. In: BPM Demo Track. vol. 1789, pp. 1–6. CEUR-WS.org (2016)
9. Cios, K., Pedrycz, W., Swiniarski, R., Kurgan, L.A.: Data mining: a knowledge discovery approach. Springer (2007)
10. Corradini, F., Muzi, C., Re, B., Rossi, L., Tiezzi, F.: MIDA: Multiple instances and data animator. In: BPM Demo. vol. 2196. CEUR-WS.org (2018)
11. Corradini, F., Muzi, C., Re, B., Rossi, L., Tiezzi, F.: Formalising and animating multiple instances in bpmn collaborations. Information Systems **103** (2022)
12. Esgin, E., Karagoz, P.: Process Profiling based Synthetic Event Log Generation. In: IC3K. vol. 1, pp. 516–524. SCITEPRESS (2019)
13. Günther, C.: Process mining in flexible environments. Ph.D. thesis, Technische Universiteit Eindhoven - Industrial Engineering and Innovation Sciences (2009)
14. Jouck, T., Depaire, B.: Generating Artificial Data for Empirical Analysis of Control-flow Discovery Algorithms: A Process Tree and Log Generator. Business and Information Systems Engineering **61**(6), 695–712 (2019)
15. Kataeva, V., Kalenkova, A.: Applying graph grammars for the generation of process models and their logs. In: Young Researchers' Colloquium on Software Engineering, vol. 8, pp. 83–87. HSE University (2014)
16. Levenshtein, V.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. vol. 10, pp. 707–710 (1966)
17. de Medeiros, A., Günther, C.: Process Mining: Using CPN Tools to Create Test Logs for Mining Algorithms. In: Practical Use of Coloured Petri Nets and CPN Tools, vol. 576, pp. 177–190. University of Aarhus (2005)
18. Meseguert, J., Montanari, U., Sassonet, V.: On the semantics of petri nets. In: Conference on Concurrency Theory. LNCS, vol. 630, pp. 286–301. Springer (1992)
19. Mitsyuk, A., Shugurov, I.S., Kalenkova, A., van der Aalst, W.: Generating event logs for high-level process models. Simulation Modelling Practice and Theory **74**, 1–16 (2017)
20. OMG: Business Process Model and Notation (BPMN V 2.0) (2011)
21. Stocker, T., Accorsi, R.: SecSy: Security-aware synthesis of process event logs. In: Enterprise Modelling and Information Systems Architectures. pp. 71–84 (2013)
22. Van Dongen, B., De Medeiros, A., Wen, L.: Process mining: Overview and outlook of petri net discovery algorithms. In: Transactions on Petri Nets and Other Models of Concurrency II, LNCS, vol. 5460, pp. 225–242. Springer (2009)
23. Weijters, A., van Der Aalst, W., De Medeiros, A.: Process mining with the heuristics miner-algorithm. In: TU/e, Tech. Rep. vol. 166, pp. 1–34 (2006)
24. Yang, H., Park, M., Cho, M., Song, M., Kim, S.: A system architecture for manufacturing process analysis based on big data and process mining techniques. In: IEEE International Conference on Big Data. pp. 1024–1029 (2014)