Generating Event Logs from Hybrid Process Models

Anti Alman[1(✉)], Fabrizio Maria Maggi[2], Marco Montali[2], and Andrey Rivkin[3]

[1] Institute of Computer Science, University of Tartu, Tartu, Estonia
anti.alman@ut.ee

[2] Free University of Bozen-Bolzano, Bolzano, Italy
{maggi,montali}@inf.unibz.it

[3] Institute of Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark
ariv@dtu.dk

Abstract. To carry on controlled experiments in process mining, it is necessary to generate event logs with specific characteristics. This has led to the development of log generation techniques in which a process model is simulated to generate an event log that is both compliant with the process model and also has certain user-defined properties (e.g., a certain number of traces, traces with certain lengths, etc.). Such techniques are available for a variety of modeling languages, both procedural and declarative. However, they are limited to simulating a single (procedural or declarative) process model at a time and do not allow simulating concurrent executions of multiple separate, but interacting, processes. In this paper, we introduce a log generation approach that takes multiple (procedural and declarative) process models (i.e., a Hybrid Business Process Representation) as input and produces an event log matching the concurrent execution of these models on the same case instances. We discuss the details of our approach and evaluate its implementation.

Keywords: Log Generation · Process Simulation · Hybrid Process Model · Data Petri net · Declare · Finite State Automaton

1 Introduction

A large number of different techniques have been developed over the past few decades for analyzing business processes, with a large portion of these techniques coming from the field of process mining [1]. The core artifact of process mining techniques is an event log, which contains sequences of events representing valid executions of a process. Processes themselves can be represented using procedural or declarative process models. The procedural paradigm is very well suited for business processes where all possible executions are relatively similar to each other. A shortcoming of the procedural paradigm is the difficulty of representing

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
J. De Weerdt and L. Pufahl (Eds.): BPM 2023 Workshops, LNBIP 492, pp. 289–301, 2024.
https://doi.org/10.1007/978-3-031-50974-2_22

processes that have high variability among all possible executions as each variation needs to be explicitly accounted for, thus leading to high complexity in the process models [14]. An alternative, the declarative paradigm, instead, assumes that every process execution is allowed by default, and then constrains these executions by defining rules to be followed throughout the process execution. This leads to process descriptions which allow for far more flexibility [2].

However, often, business processes contain both parts that can be easily represented with the procedural paradigm and parts that are better represented using declarative constraints. For example, in the healthcare domain, some clinical guidelines are procedural by nature, but declarative medical knowledge needs to be applied on top of these guidelines to deal with the heterogeneity of the patients being treated [22]. For this reason, there has been growing interest in developing hybrid approaches to combine the declarative and the procedural paradigm in order to gain the advantages of both [5].

The development and testing of such hybrid approaches can be significantly hampered by the availability of suitable event logs, especially in cases where the developed approaches need to be tested in a controlled way. In process mining, controlled experiments are, in general, carried out by using log generation techniques, which simulate an existing model to generate a corresponding event log. Many techniques are already available for different (procedural and declarative) modeling languages [6,8–10,13,18]. However, all these techniques are limited to simulating a single (procedural or declarative) process model, thus precluding simulating any form of Hybrid Business Process Representations (HBPRs) [5].

To overcome these limitations, we introduce a data-aware hybrid log generator based on the semantics for HBPRs introduced in [3,4]. In particular, this log generator can simulate the concurrent execution of any combination of declarative and procedural process models on the same case instances to generate a corresponding event log. This enables log generation for several conceptually different use cases, such as log generation from multiple process models with shared activities, simulation of a single process model defined as a set of smaller (but still overlapping) sub-models, log generation from procedural process models in the presence of additional declarative constraints and from declarative process models in the presence of additional procedural components, or simulation of multiple procedural process models connected through declarative constraints.

In this paper, we opt for Data Petri nets to represent procedural process models and a data-aware variant of the Declare language to represent declarative process specifications. These languages were selected as their models can be represented in the form of finite state automata. In fact, the log generator we present in this paper (publicly available at: https://github.com/antialman/model-interplay-loggen-code) relies on this automata-based representation of the HBPR to be simulated. Any other modeling language whose models can be translated into equivalent finite state automata can be used in our approach.

The rest of this paper is structured as follows. Section 2 discusses the core background concepts. Section 3 outlines our interpretation of concurrent execution of multiple models. Section 4 describes the log generation approach. Section 5

discusses the performance of the approach. Section 6 discusses the related work. Section 7 concludes the paper and spells out directions for future work.

2 Background

In this section, we describe the relevant background concepts of our approach. More specifically, we introduce: (i) the data-aware extensions of Petri nets and DECLARE that our approach takes as inputs, (ii) finite state automata used as an intermediary data structure, and (iii) event logs produced as output.

Data Petri Nets. As a representative of the procedural paradigm, we opt for Data Petri nets (DPNs) [12,20], which extend the traditional place-transition nets by allowing the assignment of read and write conditions (called *guards*) to the transitions of the Petri net. In our setting, DPN transitions represent the possible events of the process, DPN markings (i.e., the assignments of tokens to places) represent possible states of the process, and guards represent the relationship of process data to process control flow.

More formally, we assume a finite set $\mathcal{E} = \{\epsilon | \epsilon \in \langle n, A \rangle\}$ of event signatures, where: n is the *event* name and $A = \{a_1, \ldots, a_\ell\}$ is the set of event *attributes* (ℓ being the number of attributes). With $\mathcal{N}_\mathcal{E}$, we denote the set of all event names from $\mathcal{E}$, and with $\mathcal{A}_\mathcal{E}$, the set of all attribute names occurring in $\mathcal{E}$. Then, a *Data Petri net* D over the set $\mathcal{E}$ of is a tuple $\langle P, T, F, l, V, r, w \rangle$, where: *(i)* (P, T, F) is the Petri net graph; *(ii)* $l : T \rightarrow \mathcal{N}_\mathcal{E} \cup \{\tau\}$ is a *labeling* function (here τ denotes a *silent* transition); *(iii)* $V \subseteq \mathcal{A}_\mathcal{E}$ is the set of net's *variables*; *(iv)* $r : T \rightarrow \mathcal{G}_\mathcal{E}$ (resp., $w : T \rightarrow \mathcal{G}_\mathcal{E}$) is a *read* (resp., *write*) *guard-assignment* function, mapping every transition $t \in T$ into a read (resp., write) guard – a boolean formula $\mathcal{G}_\mathcal{E}$ whose components are atomic expressions of the form $a \odot c$, where $\odot$ is a type-specific comparison predicate and c is a constant defined in the given guard.

A DPN transition is *enabled* if and only if its read and write guards are satisfied under a given "firing mode" – a function that assigns values only to variables of the guards – and all the input places of the transition contain sufficiently many tokens to consume. To check the read guard, the firing mode function picks values currently available in the net's state, while the write guard, instead, updates the net variables to values that would satisfy the guard. When a transition is enabled, it may *fire* by consuming the necessary amount of tokens from its input places and producing the necessary amount of tokens in its output places, and by updating all the values assigned to variables in the write guard using the firing mode function. Values assigned to all other variables remain untouched.

For each DPN, we assume an initial marking (i.e., marking before any transition has fired) and a final marking (i.e., the expected marking at the end of process execution). Finally, we limit ourselves to DPNs that are 1-*bounded* and well-formed over their respective set of event signatures.

Multi-perspective Declare with Local Conditions. As a representative of the declarative paradigm, we opt for an extension of the DECLARE language [7], which we refer to as multi-perspective DECLARE with local conditions

(LMP-Declare). Both Declare and LMP-Declare describe a business process as a set of *constraints*, where each constraint defines either a control flow relation between two events (e.g., if `A` occurs then `B` must also occur) or the cardinality of a single event (e.g., `A` can occur at most twice per process execution). LMP-Declare further enriches the event references with data conditions over the event attributes, allowing the modeler to specify, for example, that `A` with $x = 2$ can occur at most twice per process execution (while leaving occurrences of `A` with any other value of x unconstrained). All Declare constraints can be represented as formulas of Linear Temporal Logic over finite traces (LTL_f) [21], and that can be extended to LMP-Declare via data abstraction techniques.

The syntax of LMP-Declare constraints is the following:

$$\Phi := \top \mid \varphi \mid \mathbf{X}\,\Phi \mid \mathbf{F}\,\Phi \mid \mathbf{G}\,\Phi \mid \Phi_1 \mathbf{U}\,\Phi_2 \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2$$

Here, φ is a boolean combination of attribute-to-constant comparisons and event variables ranging over the set of all event names from $\mathcal{E}$. Notice that the language of boolean combinations of attribute-to-constant comparisons without event variables closely resembles that of variable-to-constant conditions in [12], thus providing a good basis for combining DPNs with LMP-Declare. As in standard LTL_f, $\mathbf{X}$ denotes the *strong next* operator (which requires the existence of a next state where the inner formula holds), while $\mathbf{U}$ stands for *strong until* (which requires the right-hand formula to eventually hold, forcing the left-hand formula to hold in all intermediate states). The other two LTL_f operators are defined as $\mathbf{F}\,\Phi = \top\mathbf{U}\,\Phi$ (*eventually*) and $\mathbf{G}\,\Phi = \neg\mathbf{F}\,\neg\Phi$ (*globally*).

Deterministic Finite State Automata. In our approach, we transform the input process models into a deterministic finite state automaton (DFA), which we then use to simulate the concurrent execution of the input process models. Formally, a DFA is a labeled transition system $\mathcal{D} = \langle L, S, \delta, s_0, S_\mathrm{f}\rangle$ defined over states S and a set of labels L, having $\delta : S \times L \rightarrow S$ as the transition function, i.e., a function that, given a starting state and a label, returns the target state (if defined). $s_0 \in S$ is the initial state of $\mathcal{A}$, and $S_\mathrm{f} \subseteq S$ is the non-empty set of its accepting states ($S_\mathrm{f} \neq \emptyset$).

In our setting, the initial state of the DFA corresponds to the state of the process before any event has occurred, accepting states correspond to the final marking of a DPN and/or states in which the LMP-Declare constraints are satisfied, and the set of labels L corresponds to encoded event signatures from $\mathcal{E}$ (i.e., possible events of the business process, including the values of relevant event attributes, encoded as strings). We note that any sequence of labels, which, starting from s_0 and using the transition function δ, reaches any state in S_f, is considered to be accepted by the DFA. Furthermore, DFA are closed under the product operation $\times$ and the product of two DFAs accepts the sequences of labels accepted by both operands. In our approach, we rely on both these properties to generate event logs based on the input process models.

Event Logs. Event logs are the central artifacts in process mining and they can be seen as recordings of the executions of a business process [15]. Event logs

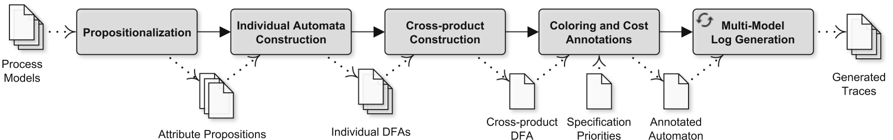


Fig. 1. Main steps and artifacts of the hybrid log generation approach.

are most commonly stored in a standardized file format called XES (eXtensible Event Stream) [16], which is structured as a set of traces, each corresponding to a single execution of a business process and consisting of an ordered set of events. Each event has a name (corresponding to the process activity executed) and may also contain an additional data payload of event attributes along with their concrete value assignments. Note that this is in line with our notion of event signatures E, which we encode into the set of DFA labels L. This allows us to use the DFA to determine sequences of events accepted by that DFA. The details of this procedure are presented in Sect. 4.

3 Interpretation of Concurrent Executions

In line with our earlier works [3,4], we interpret the concurrent execution of DPN and LMP-Declare models (for globally valid executions) as follows:

- The execution of a DPN starts from its initial marking and must reach its final marking.
- The execution of an LMP-Declare model starts with each individual constraint of that model in its initial state and must end with all of them in an accepting state.
- Events with the same name in two or more process models are considered as the same event (i.e., same-labeled events are merged for concurrent execution).
- The directly-follows relations imposed by LMP-Declare constraints (e.g., chain response) have priority over the execution of any other event that is allowed by the input models.
- Events with the same name in two or more process models can be executed whenever they are allowed by all models containing that event (unless prevented by a directly-follows relation as mentioned in the previous point).
- Each DPN variable is private to the DPN defining that variable (i.e., write guards cannot update variables of other DPNs).
- If multiple models impose conditions on the value of the same event attribute, the generated value must match the intersection of those conditions.

4 Approach

We modify and adapt specific parts of our earlier works on multi-model monitoring [3,4] to provide a data-aware hybrid log generation approach. An overview

of the main steps of the resulting approach, and its intermediary artifacts, is provided in Fig. 1. These steps are further detailed in the following subsections.

4.1 Propositionalization

As the first step, we apply data abstraction techniques to handle the data perspective of the input process models. More specifically, we first find all atomic data conditions across all models. Then, for each attribute in that set of conditions, we order all constants that this attribute is compared against. Finally, these constants, and the intervals between them, are enumerated for each attribute, thus producing attribute propositions necessary for the next step of the approach. An example of propositionalization in the case of the atomic conditions $x > 0$, $x \leq 5$, $x > 5$, and $x < 8$ is given in Table 1. Note that we also encode the attribute names ($att0$ represents attribute x in the given example).

Table 1. Propositionalization for conditions $x > 0$, $x \leq 5$, $x > 5$, and $x < 8$.

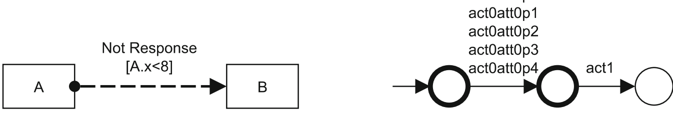
Constant		0		5		8	
Interval	$(-\infty, 0)$	$[0, 0]$	$(0, 5)$	$[5, 5]$	$(5, 8)$	$[8, 8]$	$(8, \infty)$
Interval id	p0	p1	p2	p3	p4	p5	p6
Proposition	att0p0	att0p1	att0p2	att0p3	att0p4	att0p5	att0p6

4.2 Individual DFA Construction

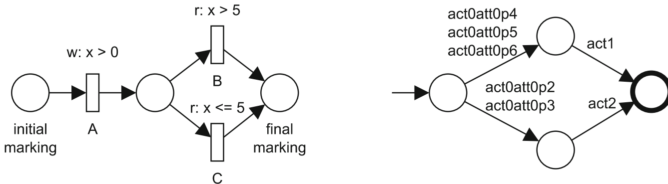
The next step after propositionalization is the construction of one DFA for each input process model using the attribute propositions.

The procedure for LMP-Declare models is fairly straightforward as it uses exactly the same set of templates as standard Declare, and equivalent LTL_f formulas are defined for all of them. These formulas can be applied in our setting by using, instead of only the event name, a conjunction of the event name with all attribute propositions matching the condition on that event. For example, event `A` with condition $x < 8$ (assuming the attribute propositions in Table 1) would be replaced with $act0att0p0 \vee act0att0p1 \vee \ldots \vee act0att0p4$, where $act0$ corresponds to event `A`. An LTL_f formula is created for each constraint in the LMP-Declare model, and a DFA of the entire model is constructed based on the conjunction of these formulas using existing automata construction techniques [23]. An example of an LMP-Declare constraint with the corresponding DFA is shown in Fig. 2(a) (thicker circles indicate accepting states).

The procedure for DPNs is more complex and for a full description we refer the reader to [4]. However, the basic idea is to construct a reachability graph of the DPN markings, where, analogously to LMP-Declare models, conjunctions of event name with the attribute propositions matching the corresponding DPN guard are used as the graph labels. An example of a DPN with the corresponding



(a) An LMP-Declare constraint and the corresponding DFA.



(b) A DPN and the corresponding DFA.

Fig. 2. DFAs with propositionalized events.

DFA is given in Fig. 2(b). At first glance, this example may seem counter intuitive as the DPN seems to have a different structure than the DFA. This difference is caused by the presence of the write guard on event `A`, which we handle by splitting the automaton into "regions" representing the value written by the guard. In Fig. 2(b), if `A` occurs with $x > 5$ then the next event must be `B` (represented by $act1$), otherwise the next event must be `C` (represented by $act2$).

Notice that the automata in Fig. 2(a) and Fig. 2(b) share some of the labels. This is because the same attribute propositions (and activity encodings) are used for both in order to allow calculating a cross-product where a single activity can result in a synchronous state change of multiple models (e.g., `A` activating the LMP-Declare constraint, while at the same time progressing the DPN).

4.3 Cross-Product Construction

After constructing the individual DFA of each model, we compute the cross-product of these DFAs using a standard automata cross-product algorithm. We do not minimize the cross-product as that would limit the possibilities of generating negative traces, i.e., traces that violate some input models.

The DFAs constructed from LMP-Declare models are used as-is. However, the DFAs constructed from DPNs are modified further. First, self loops are added to each state of the DFA such that the state of the DFA would not change if events from other models occur. Second, a non-accepting trap state is added such that this state is immediately entered whenever an event of that DPN would occur without the corresponding transition being enabled to fire.[1]

[1] Note that, for simplicity, we do not show these states in the examples provided throughout the paper.

4.4 Coloring and Cost Annotations

After computing the cross-product, we annotate it with three types of information. First, we follow the approach of colored automata [11,19] to label each cross-product state with one of four truth values for each input model, respectively indicating whether the corresponding model is *temporarily satisfied* (TS), *temporarily violated* (TV), *permanently satisfied* (PS), or *permanently violated* (PV). Second, we assume that each input model has a violation cost, which contributes to a stopping cost (*cost_curr*) attached to each state of the cross-product (i.e., the cost of stopping the execution in the given state). Third, we calculate for each state, what is the best reachable stopping cost from that state (*cost_best*) using the fixpoint procedure outlined in [4].

A fragment of the annotated cross-product is shown in Fig. 3. That fragment is based on the examples shown in Fig. 2. More specifically, it shows how the constraint in Fig. 2(a) affects the execution of the DPN shown in Fig. 2(b), while other states have been omitted.

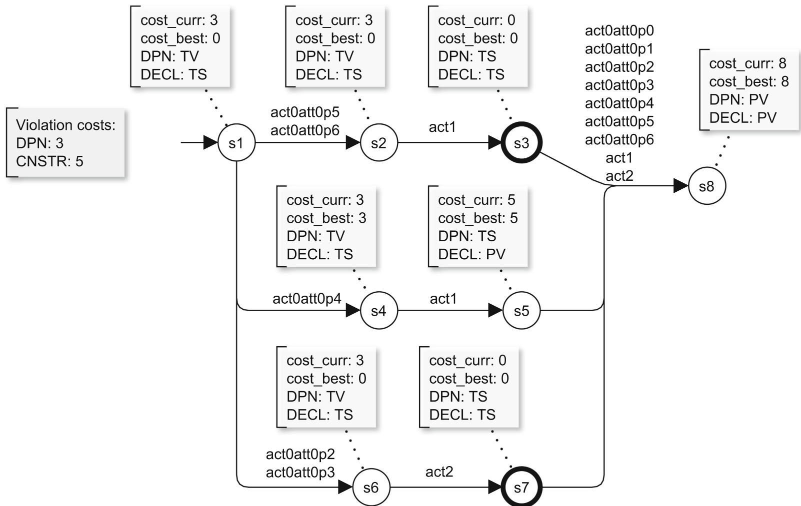
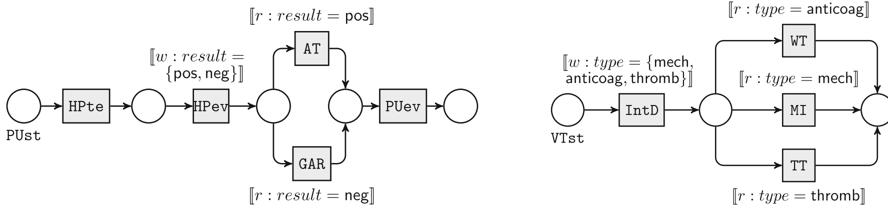


Fig. 3. Partial example of an annotated cross-product.

4.5 Multi-model Simulation

The cross-product above can be used to run multi-model simulations quite easily. The main idea is to perform semi-random walks, guided by the value of *cost_best* and translating the labels of the taken transitions back to concrete events.

For example, if the goal is to generate traces that satisfy all input process models, then, from $s1$ in Fig. 3, we can either take the transition to $s2$ or the transition to $s6$. Next, from state $s2$, we would need to take the transition to $s3$,



(a) DPN of peptic ulcer (PU) treatment.

(b) DPN of venous thromboembolism (VT) treatment.

Fig. 4. DPN models used in the performance experiments. Prefixes *r:* and *w:* distinguish read and write guards respectively.

while from $s6$ we would need to take the transition to $s7$. States $s3$ and $s7$ are accepting states in the cross-product, meaning that all sequences of transitions leading to these states correspond to traces that satisfy all input models.

We also need to translate the labels of the taken transitions back to concrete events. In the case of the transition from $s1$ to $s2$, both labels refer to activity $act0$, meaning that the event must be `A`. Both labels also contain the attribute identifier $att0$, meaning that the data payload of event `A` must contain attribute x. The propositions for $att0$ on that transition are $p5$ and $p6$, meaning that the value of attribute x must be 8 or greater, thus producing, for example, event `A[x=9]`. Applying the same translation to the other transitions, we can generate, for example, traces `<A[x=9], B>` and `<A[x=5], C>`, where the assignment of x is randomly selected from the interval (or possibly disjoint intervals) defined by the corresponding propositions.

Note that assigning x to a value in the range of (5, 8) in the above example would lead to either violating the DPN from Fig. 2(b) or the constraint from Fig. 2(a). Therefore, if we wanted to generate a trace that violates the constraint, but satisfies the DPN, we would first need to take the transition from $s1$ to $s4$ and then the transition from $s4$ to $s5$, which is possible due to not minimizing the cross-product. Generally speaking, it is possible to use the state annotations shown in Fig. 3 to generate negative traces with specific characteristics. In particular, the labels indicating the truth value of each input model (i.e., the colors) in a certain state can be used to understand which model(s) will be violated by a trace ending in that state. Furthermore, *cost_curr* and *cost_best* can be used to guide the generation of negative traces that violate the input HBPR with a certain violation cost.

5 Performance Experiments

In this section, we present the results of the performance experiments we conducted using our approach. All experiments are based on the two DPNs shown in Fig. 4. More specifically, we start with the models shown in the figure, and then iteratively increase both models by creating copies of them with renamed

Table 2. Experimental results. Columns $|P|$ and $|T|$ refer to the number of places and transitions in the corresponding input DPNs.

PU DPN		VT DPN		No. of Constraints	Cross-product time (s)	Log generation time (s)		
$\|P\|$	$\|T\|$	$\|P\|$	$\|T\|$			$n = 100$	$n = 1000$	$n = 10000$
5	5	3	4	1	0.047	0.230	1.394	11.453
9	10	5	8	2	0.180	0.356	2.185	19.004
13	15	7	12	3	0.906	0.449	3.073	27.628
17	20	9	16	4	3.440	0.562	4.040	36.713
21	25	11	20	5	11.837	0.652	4.612	45.016
25	30	13	24	6	45.955	0.724	5.413	55.460

activities and appending these copies to the original models via sequential composition. Furthermore, each pair of `AT` and `WT` is connected via a `not co-existence` constraint (without data conditions).

For each increase of the input models, we generated 100, 1000, and 10000 traces[2] with a 50:50 split between traces that satisfy all the input models and traces that violate at least one of them. The specific violations for each trace were determined randomly. All experiments were performed on a 6-core Intel i7 10850H machine with 2×16 GB of RAM and all input models are publicly available at: https://github.com/antialman/model-interplay-loggen-code.

The results of the experiments are shown in Table 2. For each set of input models, we report separately the cross-product construction time and the time needed for generating the event logs. Note that the cross-product needs to be rebuilt only if the input models are modified. In this case, we construct the cross-product once for each set of input models, and then reuse it for generating three event logs of different sizes. In general, the time (and the memory requirements) taken to create the cross-product increases rapidly as the size of the input models increases. However, models of realistic, though relatively small, sizes can be used to generate large logs (10000 traces) in a reasonable amount of time (55.4 s).

The log generation times for different log sizes (n) follow a roughly linear trend, which was expected as each trace is generated via a semi-random walk of the same cross-product. There is also a noticeable increase of the log generation time as the size of input models increases (from 11.4 s to 55.4 s for generating 10000 traces). This can be partially caused by the overall size of the cross-product being larger. Another plausible explanation for this is related to the fact that the way we extend the models results in longer traces being generated after each extension of the models (i.e., we need longer semi-random walks of the cross-product for generating traces that satisfy all input models).

Finally, we note that the main limiting factor of our approach, from the performance perspective, is memory consumption. The last test reported in Table 2 was also the largest test we could run with 32 GB of available memory. This

[2] Note that the length of the traces is constrained by the input models.

indicates that more complex models will require investigating alternatives to constructing the full cross-product, as discussed for example in [24].

6 Related Work

There are a series of approaches that use computational logic to generate log traces from declarative process models. In [9], the authors propose a tool for the synthetic generation of positive and negative event logs based on abductive logic programming for MP-Declare (an extension of Declare that allows for expressing conditions also on time and data) models. The approach in [10] relies, instead, on Answer-Set Programming for generating logs, again, from MP-Declare models. Finally, [18] proposes a log generation approach from MP-Declare models that translates the latter into Alloy programs and uses the model generation module of the Alloy Analyzer to generate traces satisfying such models. Whereas all such approaches handle trace generation from a data-aware fragment of Declare, they do not consider procedural components and neither do account for violation costs assigned to negative traces. The approach discussed in [13] relies on the traditional representation of Declare models as DFAs that, in turn, are used as string generators. Conceptually, this approach is similar to the one discussed in this paper to the point that it relies on automata-based techniques. However, it does not go beyond standard Declare specifications.

Other works propose methods for generating logs from procedural process models. One of the recent approaches in this area [8] proposes a framework for generating event logs via guided simulation of given process models. In particular, the framework is able to generate logs by taking into account specific process mining purposes such as process discovery and conformance checking. However, currently, the framework supports only procedural models (represented as BPMN or Petri nets). Similarly, the approach in [6] proposes a simulation-based log generator offering a comprehensive set of parameters for fine-tuning log generation scenarios. Finally, the CPN Tools-based approach proposed in [17] is able to generate logs from DPN models. None of the aforementioned procedural approaches has been extended to support log generation from HBPRs.

7 Conclusion

In this paper, we presented an approach to generate event logs according to the combined behavior of multiple declarative and procedural models. In its current state, we see it primarily as a tool for supporting controlled experiments in process mining. However, this instrument could also be used as a part of what-if process mining analysis pipelines where various what-if scenarios are assessed through running simulations. This use case requires additional advancements though, such as more fine grained definitions of process interplay, accounting for resource pools, and supporting durative activities. We plan to investigate the above use case as future work. Other avenues for future work include exploring

memory efficient approaches for log generation and a more extensive evaluation of the approach using real-life process models.

Acknowledgements. The work of A. Alman was supported by the Estonian Research Council grant PRG1226. F.M. Maggi was supported by the UNIBZ project CAT. M. Montali was supported by the UNIBZ project ADAPTERS and the PRIN MIUR project PINPOINT Prot. 2020FNEB27.

References

1. van der Aalst, W.M.P.: Process Mining - Data Science in Action, 2nd edn. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4
2. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: balancing between flexibility and support. Comput. Sci. Res. Dev. **23**(2), 99–113 (2009)
3. Alman, A., Maggi, F.M., Montali, M., Patrizi, F., Rivkin, A.: Multi-model monitoring framework for hybrid process specifications. In: Franch, X., Poels, G., Gailly, F., Snoeck, M. (eds.) CAiSE 2022. LNCS, vol. 13295, pp. 319–335. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-07472-1_19
4. Alman, A., Maggi, F.M., Montali, M., Patrizi, F., Rivkin, A.: Monitoring hybrid process specifications with conflict management: an automata-theoretic approach. Artif. Intell. Med. **139**, 102512 (2023)
5. Andaloussi, A.A., Burattin, A., Slaats, T., Kindler, E., Weber, B.: On the declarative paradigm in hybrid business process representations: a conceptual framework and a systematic literature study. Inf. Syst. **91**, 101505 (2020)
6. Burattin, A.: PLG2: multiperspective process randomization with online and offline simulations. In: Azevedo, L., Cabanillas, C. (eds.) Proceedings of BPM Demo Track 2016. CEUR Workshop Proceedings, vol. 1789, pp. 1–6. CEUR-WS.org (2016)
7. Burattin, A., Maggi, F.M., Sperduti, A.: Conformance checking based on multi-perspective declarative process models. Expert Syst. Appl. **65**, 194–211 (2016)
8. Burattin, A., Re, B., Rossi, L., Tiezzi, F.: A purpose-guided log generation framework. In: Di Ciccio, C., Dijkman, R., del Río Ortega, A., Rinderle-Ma, S. (eds.) BPM 2022. LNCS, vol. 13420, pp. 181–198. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-16103-2_14
9. Chesani, F., et al.: Towards the generation of the "perfect" log using abductive logic programming. In: CILC. CEUR Workshop Proceedings, vol. 2396, pp. 179–192. CEUR-WS.org (2019)
10. Chiariello, F., Maggi, F.M., Patrizi, F.: ASP-based declarative process mining. In: AAAI, pp. 5539–5547. AAAI Press (2022)
11. De Giacomo, G., De Masellis, R., Grasso, M., Maggi, F.M., Montali, M.: Monitoring business metaconstraints based on LTL and LDL for finite traces. In: Sadiq, S., Soffer, P., Völzer, H. (eds.) BPM 2014. LNCS, vol. 8659, pp. 1–17. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10172-9_1
12. de Leoni, M., Felli, P., Montali, M.: A holistic approach for soundness verification of decision-aware process models. In: Trujillo, J.C., et al. (eds.) ER 2018. LNCS, vol. 11157, pp. 219–235. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00847-5_17
13. Di Ciccio, C., Bernardi, M.L., Cimitile, M., Maggi, F.M.: Generating event logs through the simulation of declare models. In: Barjis, J., Pergl, R., Babkin, E. (eds.) EOMAS 2015. LNBIP, vol. 231, pp. 20–36. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24626-0_2

14. Di Ciccio, C., Marrella, A., Russo, A.: Knowledge-intensive processes: characteristics, requirements and analysis of contemporary approaches. J. Data Semant. **4**(1), 29–57 (2015)
15. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, 2nd edn. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-56509-4
16. Gunther, C.W., Verbeek, H.M.W.: XES - standard definition, BPM reports, vol. 1409. BPMcenter.org (2014)
17. Li, G., de Carvalho, R.M., van der Aalst, W.M.P.: A model-based framework to automatically generate semi-real data for evaluating data analysis techniques. In: Filipe, J., Smialek, M., Brodsky, A., Hammoudi, S. (eds.) Proceedings of ICEIS 2019, pp. 213–220. SciTePress (2019)
18. Maggi, F.M., Marrella, A., Patrizi, F., Skydanienko, V.: Data-aware declarative process mining with SAT. ACM Trans. Intell. Syst. Technol. (2023)
19. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring business constraints with linear temporal logic: an approach based on colored automata. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 132–147. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23059-2_13
20. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P.: Balanced multi-perspective checking of process conformance. Computing **98**(4), 407–437 (2016)
21. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographiess. ACM Trans. Web **4**(1), 3:1–3:62 (2010)
22. Piovesan, L., Terenziani, P., Dupré, D.T.: Temporal conformance analysis and explanation on comorbid patients. In: HEALTHINF, pp. 17–26. SciTePress (2018)
23. Westergaard, M.: Better algorithms for analyzing and enacting declarative workflow languages using LTL. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 83–98. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23059-2_10
24. Westergaard, M., Slaats, T.: Mixing paradigms for more comprehensible models. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 283–290. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40176-3_24