Bus Inf Syst Eng 61(6):695–712 (2019)
https://doi.org/10.1007/s12599-018-0541-5

RESEARCH PAPER

Generating Artificial Data for Empirical Analysis of Control-flow Discovery Algorithms

A Process Tree and Log Generator

Toon Jouck · Benoît Depaire

Received: 18 January 2017 / Accepted: 23 March 2018 / Published online: 23 April 2018

Abstract Within the process mining domain, research on comparing control-flow (CF) discovery techniques has gained importance. A crucial building block of empirical analysis of CF discovery techniques is obtaining the appropriate evaluation data. Currently, there is no answer to the question of how to collect such evaluation data. The paper introduces a methodology for generating artificial event data (GED) and an implementation called the Process Tree and Log Generator. The GED methodology and its implementation provide users with full control over the characteristics of the generated event data and an integration within the ProM framework. Unlike existing approaches, there is no tradeoff between including long-term dependencies and soundness of the process. The contributions of the paper provide a solution for a necessary step in the empirical analysis of CF discovery algorithms.

Keywords Artificial event logs · Process discovery · Empirical analysis

1 Introduction

Process mining is the research domain focused on extracting knowledge from process execution logs, commonly referred to as event logs (van der Aalst 2016; Dumas et al. 2013). Most attention within process mining has been paid to a group of techniques aimed at control-flow (CF) discovery whose goal it is to discover the process control-flow directly from an event log. Over the past 15 years, researchers have developed a multitude of algorithms for CF discovery (for an overview see van der Aalst 2016; De Weerdt et al. 2012). In the early days researchers developed algorithms for discovering specific process constructs, while recently, new algorithms focus on outperforming existing algorithms in terms of certain quality measures. This shift has led to an increased importance of research on comparing such algorithms (Rozinat et al. 2007; De Weerdt et al. 2012; vanden Broucke et al. 2014; Wang et al. 2012; Weber et al. 2013).

The framework introduced by Rozinat et al. (2007) describes an empirical evaluation method to compare CF discovery techniques. Such an evaluation requires large amounts of appropriate data (models and event logs) as input for empirical analysis. Yet, which data is appropriate to use for comparing CF discovery algorithms? How to collect such data? These questions have received little to no explicit attention, despite their many challenges.

To illustrate the problem, consider the following example: a researcher investigates which algorithm performs best for rediscovering processes with structured loops and long-term (LT) dependencies, algorithm x or y? An empirical comparison requires logs that guarantee such behavior while controlling for other behavior. Therefore, the researcher needs a method to generate these logs which guarantees a correct experimental design.

This paper advocates the use of artificial event logs rather than real event logs for empirical evaluation. First of all, the process population characteristics of a real event log are unknown because a reference model is lacking. Secondly, the number of real event logs is limited. However, in order to draw statistically significant conclusions, large amounts of data sets are needed. Artificially

Accepted after two revisions by Prof. Dr. Bichler.

T. Jouck (✉) · Prof. Dr. B. Depaire
Faculty of Business Economics, Hasselt University, Agoralaan Bldg D, 3590 Diepenbeek, Belgium
e-mail: toon.jouck@uhasselt.be

generated event data can overcome both of these limitations and therefore are more appropriate for tackling the challenge of collecting event data for empirical analysis of CF discovery techniques.

None of the current artificial event data approaches presents a general methodology of how to generate process models and event logs to empirically evaluate process discovery algorithms. However, this is necessary to ensure a correct experimental design which in turn guarantees statistically valid conclusions of the empirical analysis. Moreover, in current approaches a tradeoff exists between the inclusion of LT dependencies and soundness of the final model. On one hand, approaches using block-structured models guarantee sound models but naturally cannot handle LT dependencies. On the other hand, approaches with models that include long-term dependencies cannot guarantee soundness. However, block-structured models impose a rather restrictive assumption (van der Aalst 2016), while soundness ensures that simulation of a model (i.e., the generation of an event log) cannot get stuck in a deadlock or livelock.

To overcome these issues, this paper introduces a methodology and implementation for generating random artificial process models and event logs to enable empirical CF discovery analysis. The objective fits into the design science framework as it aims at the scientific study and creation of artifacts with the goal of solving practical problems of general interest (Johannesson and Perjons 2014). Design science methodology defines four fundamental steps: define requirements, design and develop artifact(s), demonstrate artifacts, and evaluate artifacts. This paper is structured accordingly. It makes the following contributions:

- A general methodology for generating random artificial process models and event logs (Sect. 2).
- Implementation of the methodology for generating random sound process models with LT dependencies and corresponding event logs (Sect. 4).

Section 5 demonstrates and evaluates the generated artifacts. Section 6 summarizes the conclusions of the paper.

2 GED Methodology Versus Related Work

Approaches for generating artificial event data have already been introduced by Burattin (2015), Jin et al. (2011), van Hee and Liu (2010), Kataeva et al. (2014). Each of these approaches focus on the algorithms and implementation of generating artificial models and event logs. However, none of the existing approaches presents a methodology of how to generate event data for empirically evaluating CF discovery techniques. Such a methodology, nonetheless, is an essential starting point to ensure that the empirical analysis has a sound experimental design that guarantees valid statistical claims.

To fill this gap, the starting point of the paper is a new methodology for Generating artificial Event Data (GED) for process discovery evaluation. This methodology (illustrated in Fig. 1) consolidates concepts of experimental design in statistics with existing process mining research methodology. To our knowledge, this is the first time that a methodology combines the ideas of those two research areas. The GED methodology is the foundation from which specific requirements for our artifacts are derived.

2.1 GED Methodology

The GED methodology uses a hierarchical experimental design (Box et al. 2005) for the generated event data. Figure 2 illustrates this design: the first level comprises the process model population (hereafter called model population), the second level a random sample of process models, and the third level a random sample of event logs generated from the models in the second level. This structure gives researchers full control over the control-flow behavior in the generated event data. Additionally, it enables the researcher to generalize findings from the event logs to a known model population.

The GED methodology starts by defining the model population. A model population specifies the control-flow patterns and their probabilities. Examples of such patterns are the workflow control-flow patterns (WCP), identified by Russell et al. (2006), which represent process behavior common to all real business processes. A probability distribution is assigned to each pattern such that the sample (drawn in the second step) contains random models from the population. Each model in the sample will then be simulated into a set of event logs while setting parameters to control the number of traces and the amount of noise. This set of logs forms a sample of all possible logs produced by the model population.

The last two steps of the GED methodology are adopted from existing process mining methodology (see vanden Broucke et al. 2014; Weber et al. 2013; Wen et al. 2007; de Medeiros et al. 2007). In contrast to existing approaches in which researchers typically created models by hand in an ad hoc manner, GED generates models which are random observations from a model population. This allows researchers to generalize their results to a pre-defined population.

One possible use case for the GED methodology is the performance comparison of CF discovery algorithms in terms of model quality. In this case a researcher needs to define populations with an extended set of control-flow patterns. If only a limited set of basic patterns were

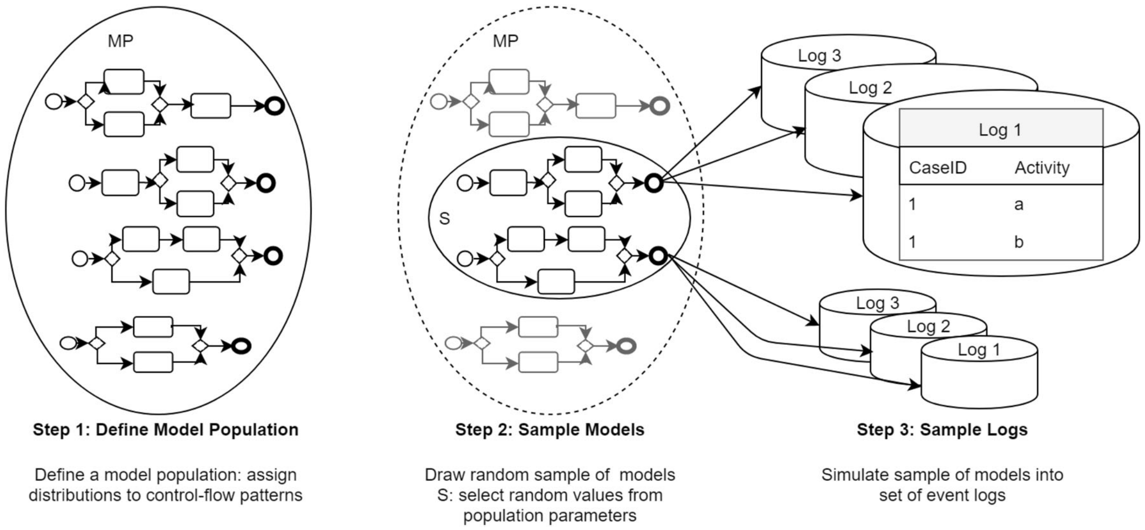

MP
MP
S
Log 3
Log 2
Log 1
CaseID
Activity
1
a
1
b
Log 3
Log 2
Log 1
Step 1: Define Model Population
Define a model population: assign distributions to control-flow patterns
Step 2: Sample Models
Draw random sample of models S: select random values from population parameters
Step 3: Sample Logs
Simulate sample of models into set of event logs

Fig. 1 The GED methodology for generating artificial event data

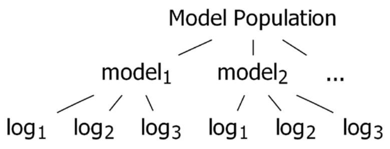


Fig. 2 General methodology: a hierarchical model

available, the simplicity of the event data could bias the comparison results. A second use case for GED is the goal to understand the effect of specific control-flow behavior on algorithm performance. This use case requires full control over all possible control-flow patterns in the generated models to enable causal analysis.

Based on these use cases, more specific requirements for the GED methodology implementation are derived (see leftmost column of Table 1). The first group of requirements regards the *full control* over the control-flow behavior in the generated process models (control-flow patterns) and event logs (log characteristics). A multitude of evaluation studies vanden Broucke et al. (2014), de Medeiros et al. (2007), van Dongen et al. (2009) assessed discovery algorithms using an extensive set of control-flow patterns. This set includes the basic WCP (Russell et al. 2006): sequence (WCP-1), parallelism (WCP-2 and 3), exclusive choice (WCP-4 and 5), ‘or’ (WCP-6 and 7) and structured loops (WCP-21). Besides the basic patterns, the set also covers the complex constructs invisible (skipping) activities, duplicate activities and LT dependencies. De Medeiros and vanden Broucke et al. (2014) also studied the effect of log characteristics, i.e., number of traces, noise, and infrequent behavior.[1] Consequently, the implementation of the GED methodology should support control over all these patterns and characteristics.

Additionally, the soundness was added as a requirement for each generated model. This ensures that the produced model can never cause a deadlock during the simulation. A simulator allowing for unsound models requires the detection of the violation and the repair of that violation which is far from trivial (Buijs 2014).

The second group of requirements relates to *randomness*. In order to generalize findings from event logs to the model population, the event logs should be random samples to avoid biased conclusions. To be more specific, the implementation should allow to draw a random sample of models from the model population. In the next step, the implementation should support the simulation of a random sample of logs from the sample of models.

The third and last group of requirements specifies the *formats* of the generated event data and *integration* with process mining tools. A discovery evaluation experiment can exploit the extensive set of algorithms and conformance checking techniques in the ProM framework (Verbeek et al. 2011). This framework uses the XES standard (Verbeek et al. 2011) for event logs and supports different XML-based formats for process models. Therefore, it is important that the implementation of the GED methodology produces models and logs in these standard formats. An additional advantage would be the integration within the ProM framework (Verbeek et al. 2011) to enable automated experiments.

Each of the requirements are listed in the leftmost column of Table 1, grouped by category: full control, randomness, and standard formats. The use of the GED

[1] Noise is defined in this paper as incorrect behavior in the log (see Sect. 4.4).

Table 1 Evaluating existing implementations on the requirements of GED implementation

	PLG2 (Burattin 2015)	GraphGrammar (Kataeva et al. 2014)	BeehiveZ (Jin et al. 2011)	TestBed (van Hee and Liu 2010)
R1 full control				
Number of activities		✓	✓	
Sequence (WCP-1)	✓	✓		✓
Parallel (WCP-2-3)	✓	✓		✓
Choice (WCP-4-5)	✓	✓		✓
Loop (WCP-21)	✓			✓
Or (WCP-6-7)				
Silent (skipping) activities	✓			
Reoccurring (duplicate) activities				
Long-term (LT) dependencies				✓[b]
Infrequent paths	✓			
Soundness	✓	✓		[b]
No. traces	✓	✓	✓	
Noise	✓		✓	
R2 randomness				
Random generation	✓	✓	✓	✓
R3 standard formats				
Models	✓		✓	✓
Logs	✓		✓	
ProM integration	✓[a]			✓

[a]PLG2 (Burattin 2015) is only available as a standalone tool, but the older PLG (Burattin and Sperduti 2011) is implemented in ProM with all the indicated requirements

[b]The approach allows to add ’arc bridges’ to create non-free choice constructs, yet it does not guarantee sound models

methodology for artificial event data needs an implementation that supports all these requirements.

2.2 Evaluation of Related Work

We have evaluated the existing implementations for generating event data against the requirements stated above. The results in (Table 1) show that none of the existing tools fulfills all the requirements. PLG2 (Burattin 2015) is the most mature tool, but is limited to block-structured process models, i.e., the models cannot contain LT dependencies, which imposes a rather restrictive assumption (van der Aalst 2016). The approach using GraphGrammar (Kataeva et al. 2014) only allows for generating and simulating rather simplistic process models. The BeehiveZ tool (Jin et al. 2011) gives users limited control over the control-flow constructs in the generated models as users can only pick a class of process models from which a random sample is drawn and simulated. Finally, the TestBed tool (van Hee and Liu 2010) does not include a simulator, but allows for LT dependencies. However, to model LT dependencies it uses non-free choice constructs which do not guarantee soundness of the produced models. An alternative solution to guarantee soundness would be as follows: first, generate a subclass of Workflow nets (called Jackson nets (van Hee and Liu 2010) that are always sound, then extend these models with non-free choice constructs (NFC) to introduce LT dependencies. Each time a NFC is added, check for soundness, if there is a violation, revert the NFC and try another. However, deciding soundness may be intractable for complex nets (van der Aalst 1998) and therefore this solution is insufficient. Moreover, the randomness of the generated models, another requirement of the GED methodology, is possibly violated as some random sound models are excluded if they require a series of bridge rules which first make the model unsound and later on make the model sound again.

Other approaches for artificial event data generation such as SecSy (Stocker and Accorsi 2013), CPN Tools (Jensen et al. 2007) and GENA log generator (Mitsyuk et al. 2017) only focus on simulation of an event log given a process model as input. These approaches are less relevant as the paper focuses more on model than on log generation.

As none of the current tools meets all the requirements, this paper introduces an implementation of the GED methodology that conforms to all the requirements. In this paper we will mostly focus on the challenge of solving the trade-off between including LT dependencies while ensuring soundness of the generated models.

3 Preliminaries

The implementation of the GED methodology uses process trees (van der Aalst et al. 2012; Buijs 2014) for modeling the randomly generated processes. A first advantage of process trees is that each tree is inherently sound (van der Aalst et al. 2012). The soundness property guarantees that the simulation of a tree, in the simulation step of the methodology, can never deadlock. Another advantage is that process trees can easily be built in a stepwise manner using the patterns defined in the population as building blocks. Definition 1 formalizes a process tree *PT*(*N*, *r*, *m*, *c*, *p*, *s*, *b*) used in the remainder of this paper. It extends the definition by Buijs (2014) with a parent (*p*) and a probability mapping function (*b*).

Definition 1 (*Process tree*) Let $A \subseteq \mathcal{A}$ be a finite set of activities and PT be a tree: $PT = (N, r, m, c, p, s, b)$, where:

- N is a non-empty set of nodes consisting of operator (N_O) and leaf nodes (N_L) such that: $N_O \cap N_L = \emptyset$
- $r \in N_O$ is the root node of the tree
- $O = \{\rightarrow, \times, \wedge, \circlearrowleft^k, \vee\}$ are the base patterns: 'sequence','choice','parallel','loop' and 'or'.
- $m : N \rightarrow A \cup O$ is a mapping function mapping each node to an operator or activity, with τ representing a silent activity:
$$m(n) = \begin{cases} a \in A \cup \{\tau\}, & \text{if } n \in N_L. \\ o \in O, & \text{if } n \in N_O. \end{cases}$$
- Let N^* be the set of all finite sequences over N then $c : N \rightarrow N^*$ is the child-relation function:
  $c(n) = \langle\rangle$ if $n \in N_L$
  $c(n) \in N^*$ if $n \in N_O$
  such that
  - each node except the root node has exactly one parent:
  $$\forall n \in N \backslash \{r\} : \exists p \in N_O : n \in c(p) \wedge \nexists q \in N_O : p \neq q \wedge n \in c(q);$$
  - the root node has no parent: $\nexists n \in N : r \in c(n)$;
  - each node appears only once in the list of children of its parent: $\forall n \in N : \forall_{1 \leq i < j \leq |c(n)|} : c(n)_i \neq c(n)_j$;
  - a node with a loop operator type has exactly three children such that the first child is always executed first, the second child is executed maximum $k \in \mathbb{N}$ times, each time followed by the first child, and finally the third child is executed once: $\forall n \in N : (m(n) = \circlearrowleft^k) \Rightarrow |c(n)| = 3$.
- $p : N \rightarrow N$ is the parent relation function: $p(n) = q \Leftrightarrow n \in c(q)$
- each node has a probability of being chosen: $b : N \rightarrow [0, 1]$ is a mapping function mapping a probability to each node n:
$$b(n) = \begin{cases} 1, & \text{if } p(n) \notin N_\times \\ \in [0,1], & \text{such that } \sum_{k}^{k \in c(p(n))} b(k) = 1 \text{ if p(n)} \in \mathrm{N}_\times. \end{cases}$$
- Let N^* be the set of all finite sequences over N then $s : N \rightarrow N^*$ is the subtree function, returning all nodes of n in a pre-order:
$$s(n) = \begin{cases} n, & \text{if } n \in N_L. \\ n \cdot s(c(n)_1) \cdot \ldots \cdot s(c(n)_{|c(n)|}), & \text{if } n \in N_O. \end{cases}$$
- A node $n \in N$ can be denoted in shorthand as follows: $n = t\langle n_1, \ldots, n_k \rangle$ where $t = m(n)$ and $\langle n_1, \ldots, n_k \rangle = c(n)$.

Figure 3 shows an example process tree $PT_1 = \rightarrow (\times(a, b), c, \wedge(d, e))$ that represents a simple process that starts with a choice between activities *a* and *b*, followed by activity *c*, and then followed by activity *d* and *e* in parallel.

4 Process Tree and Log Generator

This section describes the implementation of the GED methodology: (1) how to specify a model population, (2) how to draw a sample of process trees from that population, (3) how to add random LT dependencies to a process tree, and (4) how to simulate a tree into an event log. The requirements stated in Sect. 2 are taken as the starting point for all steps.

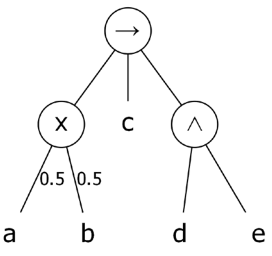


Fig. 3 $PT_1 = \rightarrow (\times(a, b), c, \wedge(d, e))$

4.1 Define a Model Population

The first step of the GED methodology is the definition of a model population. In this step the user defines the building blocks, i.e., control-flow patterns, of which the models in the population consist. The full control requirement category in Sect. 2 listed an extensive set of control-flow patterns a researcher wants to control during discovery algorithm evaluation (listed in the first column of Table 2).

In the implementation of the methodology, the model population consists of process trees. Each process tree consists of operator and leaf nodes (see Definition 1). An operator node represents one of the basic workflow control-flow patterns: 'sequence' ($\rightarrow$), 'choice' ($\times$), 'parallel' ($\wedge$), 'loop' ($\circlearrowleft^k$) and 'or' ($\vee$). A leaf node represents an activity which can be a visible activity ($a \in A$) or a silent activity (τ).

Our method requires users to specify six probability distributions to define a tree population. First, the user assigns a triangular distribution for the number of visible activities to control the size of the trees. A triangular distribution is characterized by a lower limit, a mode and an upper limit: y $\sim$ triangular(minimum,mode,maximum). As a result, all trees in the population will have a number of visible activities $y = |\{n \in N_L | m(n) \in A\}|$ between the lower and upper limit with the mode as most likely value.

Secondly, the frequency of the operator types 'sequence' ($\rightarrow$), 'choice' ($\times$), 'parallel' ($\wedge$), 'loop' ($\circlearrowleft$) and 'or' ($\vee$) in a tree is defined by a categorical distribution. As a result, each of these operator types has a fixed probability: $\Pi^{\rightarrow}, \Pi^{\wedge}, \Pi^{\times}, \Pi^{\vee}, \Pi^{\circlearrowleft}$. Together the probabilities of these basic patterns should always sum to one. Therefore, every tree in the population has at least one operator node to rule out an overly simplistic tree with only one leaf node.

Finally, each of the more complex patterns are assigned to a binomial distribution. The number of silent activities depends on the probability Π^{τ} to add a silent activity to a 'choice' or 'loop' node. The number of reoccurring activities is determined by the probability to duplicate a visible activity Π^{Re}. The number of LT dependencies is subject to the likelihood Π^{Lt} of inserting a dependency between activities in 'choice' nodes. Finally, the number of 'choices' with infrequent outgoing path(s) depends on the probability Π^{In}.

Definition 2 formalizes a model population *MP* used in the remainder of this paper.

Definition 2 (*Model Population*) A model population is defined as $MP =$ (minimumVisibleAct, modeVisibleAct, maxVisibleAct, Π^{Base},Π^{τ},Π^{Re},Π^{Lt},Π^{In}) such that:

- The number of visible activities $y \sim triangular$ (minimumVisibleAct, modeVisibleAct, maxVisibleAct)
- The type of an operator node $n \in N_O \sim Categorical(\Pi^{Base})$ with
  $$\Pi^{Base} = \{\Pi^{\rightarrow}, \Pi^{\wedge}, \Pi^{\times}, \Pi^{\vee}, \Pi^{\circlearrowleft}\}$$
- The number of silent activities $\sim Binomial(|\{N_{\times} \cup N_{\circlearrowleft}\}|, \Pi^{\tau})$
- The number of duplicated visible activities $\sim Binomial(y, \Pi^{Re})$ with y the number of visible activities
- The number of LT dependencies $\sim Binomial(\Delta, \Pi^{Lt})$ with Δ the total number of possible dependencies
- The number of choice nodes with infrequent paths $\sim Binomial(|N_{\times}|, \Pi^{In})$

Table 2 Probability settings of control-flow patterns

Parameter	Setting
Number of visible activities	(min,mode,max)
Sequence ($\Pi^{\rightarrow}$) (WCP-1)	$\in [0, 1]$
Parallel ($\Pi^{\wedge}$) (WCP-2/3)	$\in [0, 1]$
Choice ($\Pi^{\times}$) (WCP-4/5)	$\in [0, 1]$
Loop ($\Pi^{\circlearrowleft}$) (WCP-21)	$\in [0, 1]$
Or ($\Pi^{\vee}$) (WCP-6/7)	$\in [0, 1]$
Silent activities (Π^{τ})	$\in [0, 1]$
Reoccurring activities (Π^{Re})	$\in [0, 0.5]$
Long-term dependencies (Π^{Lt})	$\in [0, 1]$
Infrequent paths (Π^{In})	$\in [0, 1]$

4.2 Sample Models

The definition of the model population enables the second step of the GED methodology: draw a random sample of models from the population. The implementation of this step uses a process tree generating algorithm illustrated in Fig. 4.

The tree building algorithm in Fig. 4 builds a random process tree *PT* given a model population *MP*. It starts by drawing a random value *y* from the distribution of activities to decide how large the tree will grow in terms of visible activities. After that, the algorithm adds nodes to the tree for as long as there are activities left to incorporate ($\#act < y$). In each iteration the algorithm selects a random visible leaf node (or the root node in case the tree has no nodes yet) and replaces this node with an operator node based on the probabilities in Π^{Base}.[2] Then, the algorithm adds leaf nodes to the assigned operator: a loop node always has three leaf nodes, all the other operators get two

[2] Invisible activities are endpoints in the tree and hence are never selected to be replaced.

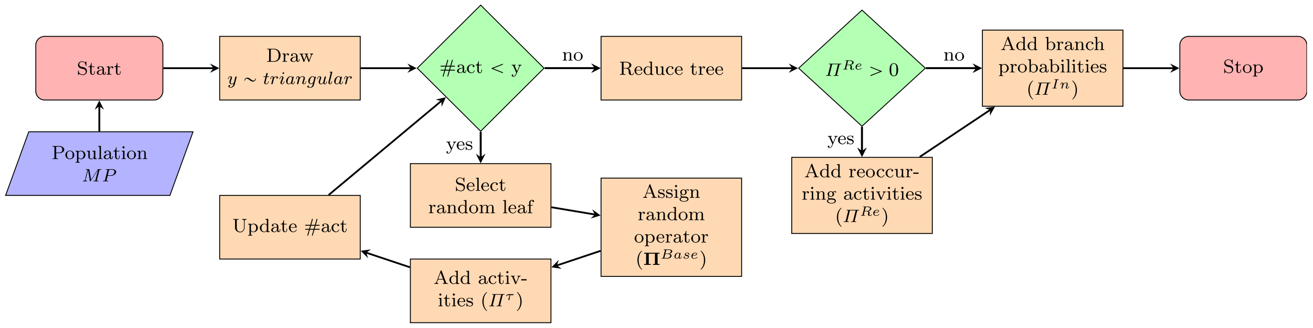


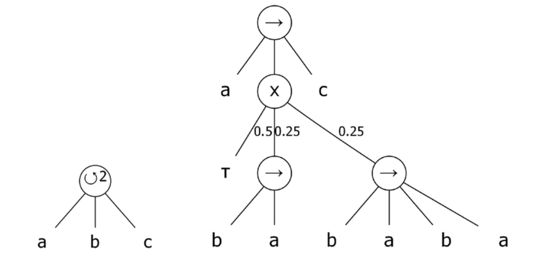
Fig. 4 Flowchart of tree building algorithm

leaf nodes. If the operator is of type choice or loop, one of the added leaf nodes can be an invisible activity based on the probability Π^{τ}. The next step updates the number of visible activities $\#act$ in the tree. After all activities are added ($\#act = y$), the tree is reduced. This step merges parent and child nodes if they have the same operator type except for loops.[3] As a result, the reduced tree is not limited to operators with only two children. The next step of the algorithm duplicates the labels of leaf nodes based on the probability Π^{Re}. Finally, the algorithm assigns either equal or unequal branch probabilities to each choice node based on the probability Π^{In}.

This paper adopts the process tree operator semantics from Buijs (2014). For each operator there exists a trace equivalent Petri net translation. A process trees generated by the above algorithm is free-choice and therefore does not contain LT dependencies. Section 4.3 presents a method to add random LT dependencies to a given process tree.

4.3 Adding Long-term Dependencies

Process trees, as generated in the previous step, are block-structured models, i.e., all trees can be decomposed into blocks with single entry and single exit points. As a result, all dependencies in a tree are local, i.e., there are no LT dependencies. Yet, such dependencies are part of the full control requirement of the GED methodology. Previous approaches that generate models with LT dependencies do not guarantee soundness, another requirement of the GED methodology. Therefore, this paper proposes an approach to incorporate random LT dependencies in a given tree resulting in a so called 'unfolded choice tree' which is always sound (see Algorithm 1). The paper adopts the definition of LT dependencies of Buijs (2014): "choices that depend on decisions made earlier in the process". It focuses on decisions represented as exclusive choices



(a) Original loop $\circlearrowleft^2(a, b, c)$ **(b)** Unfolded loop with maximum two repetitions

Fig. 5 Illustration of the loop unfolding step

(WCP-4 and 5), as such the considered LT dependencies correspond to the non-free-choice constructs in cases a, e, f and g of Fig. 5 in (Wen et al. 2007). To our knowledge, this approach is the first to extend process trees with LT dependencies.

As an example, consider the process tree PT_2 illustrated in Fig. 6a. Tree PT_2 has a 'sequence' operator as root node with several 'choice' nodes (choices) as descendants.[4] PT_2 contains no LT dependencies, e.g., if activity a was chosen in $\times(a, b)$, then this decision would not affect the choice between f and g in $\times(f, g)$ later in the process. The proposed approach will allow to incorporate LT dependencies to allow for dependencies between choices. Consider for example a dependency between activities in choices $\times(a, b)$ and $\times(f, g)$: if a is chosen, then f cannot be chosen later on.

The following subsections will describe the two steps of the proposed approach to insert LT dependencies: a tree preparation step followed by an insertion step.

4.3.1 Preparing the Tree for Long-term Dependencies

The first step of the approach to insert LT dependencies is a preparation step. A LT dependency limits the choice behavior of one choice based on what happened in

[3] Reducing parent and child loop nodes could cause the parent loop node to have more than three children.

[4] A descendant is a node reachable by repeatedly going from parent to child.

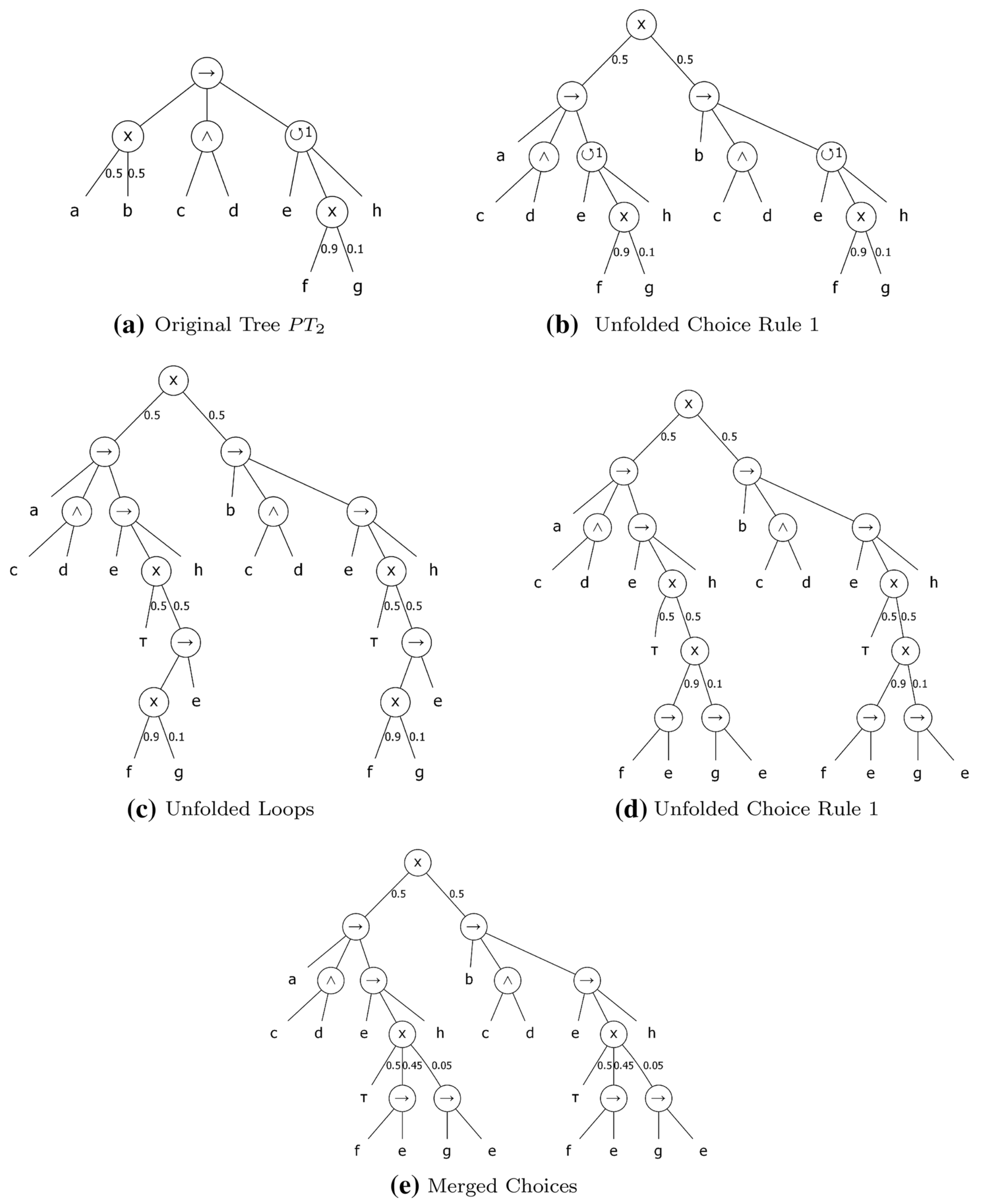


Fig. 6 Unfolding of tree PT_2

(an)other choice(s). For example, a dependency between activity a and f in Fig. 6a limits the behavior in choice $\times(f, g)$, i.e., if a happens, then f cannot be chosen in $\times(f, g)$. As such, a LT dependency forbids behavior in a combination of choices of a tree.

To insert LT dependencies in a process tree, one needs combinations of choice behavior. However, a process tree in its normal form does not display such combinations. Therefore, the proposed approach first transforms the given tree PT into a trace equivalent tree called the unfolded choice tree using duplication of activity labels. The transformed tree, denoted as $PT^{\times}$, contains only one choice which is the root node r'. Each branch (subtree) under the root r' contains a combination of choice behavior in the original tree. As such, the choice at the root r' represents all choices in the original tree. At the same time, $PT^{\times}$ is still block-structured and thus sound (see proof of Theorem 1).

Lines 10–18 of Algorithm 1 describe how to unfold the original tree PT into $PT^{\times}$ in a recursive way using the transformation rules in Definition 3. Each time, take the deepest choice in the tree and apply a transformation rule in Definition 3 to move it closer to the root node. Notice that there is no transformation rule for a loop node with a choice as the first or second child as a direct unfolding of such a choice would make $PT^{\times}$ not trace equivalent to PT.[5] Therefore, the user can decide if such choices in loops are unfolded. Not unfolding these first and second child choices will exclude them from the generated LT dependencies. If a user chooses to unfold the choices in the first or second child of the loop, then this requires a special unfolding step for that particular loop.

Definition 1 specifies that a loop node has exactly three children such that the first child node is always executed first, the second node is executed maximum k times, each time followed by the first child node, and finally the third child node is executed to conclude. Because k is a finite number, one can unfold the bounded loop into a trace equivalent structure of $\rightarrow$ and $\times$ nodes as illustrated in Fig. 5 with $k = 2$. The bounded loop can be justified by accepting a so-called fairness assumption by van der Aalst (1998): "soundness and strong fairness means that each process instance will eventually terminate correctly". The user can specify the number k, i.e., the maximum times a loop can repeat. After the loop unfolding, the resulting choices can be unfolded again with the rules in Definition 3.

When applying the transformation rules in Definition 3 the branching probabilities of the children of the original choice move to the children of the new unfolded choice. The loop unfolding results in a choice node with as children the number of loop repetitions. The probability of these repetitions is defined using a categorical distribution: $\Pi^{Repetitions} = \{\Pi^0, \Pi^1, \ldots, \Pi^{k-1}, \Pi^k\} : \Pi^i = 0.5^{i+1}\ \forall i \in [0, k-2]$ and $\Pi^i = 0.5^k\ \forall i \in [k-1, k]$, where Π^i is the probability of i repetitions and k the maximum number of repetitions. As such this distribution is equivalent to the behavior of a bounded loop with a probability of 50% to do a loop iteration and a probability of 50% to exit the loop.

Definition 3 (*Unfolded Choice Tree*) A given tree $PT = (N, r, m, c, p, s, b)$ with at least one choice block, i.e., $|\{\times_i | \times_i \in N_O\}| \geq 1$, can be transformed to the unfolded choice tree form $PT^{\times}$ using the following rules:

1. $\rightarrow(\times(\ldots_1, \ldots_2), \ldots_3) = \times(\rightarrow(\ldots_1, \ldots_3), \rightarrow(\ldots_2, \ldots_3))$
2. $\times(\times(\ldots_1, \ldots_2), \ldots_3) = \times(\ldots_1, \ldots_2, \ldots_3)$
3. $\wedge(\times(\ldots_1, \ldots_2), \ldots_3) = \times(\wedge(\ldots_1, \ldots_3), \wedge(\ldots_2, \ldots_3))$
4. $\circlearrowleft^k(\ldots_1, \ldots_2, \times(\ldots_3, \ldots_4)) = \times(\circlearrowleft^k(\ldots_1, \ldots_2, \ldots_3), \circlearrowleft^k(\ldots_1, \ldots_2, \ldots_4))$
5. $\vee(\times(\ldots_1, \ldots_2), \ldots_3) = \times(\vee(\ldots_1, \ldots_3), \vee(\ldots_2, \ldots_3))$

The branching probabilities assigned to each of the children of a choice node $\times_i$ by the mapping function $b(c(\times_i)_j)$ in PT are transferred to the new choice $\times_i'$ in $PT^{\times}$ each time a rule is applied:

- if $p(\times_i) \in \{N_{\rightarrow} \cup N_{\wedge} \cup N_{\vee} \cup N_{\circlearrowleft}\}$, then the probabilities move up with the $\times_i$ operator: $b(c(\times_i')_j) = b(c(\times_i)_j)$
- if $p(\times_i) \in N_{\times}$, then the branching probabilities of both choice nodes are multiplied when merging: $\times_p(\times_i(\ldots_1, \ldots_2), \ldots_3) = \times_{pi}'(\ldots_1, \ldots_2, \ldots_3)$, then the probabilities of $\times_{pi}'$ are:
  - $b(c(\times_{pi}')_1) = b(c(\times_p)_1) \cdot b(c(\times_i)_1)$
  - $b(c(\times_{pi}')_2) = b(c(\times_p)_1) \cdot b(c(\times_i)_2)$
  - $b(c(\times_{pi}')_3) = b(c(\times_p)_3)$

To illustrate the tree transformation, consider the tree PT_2 in Fig. 6a. First, select the deepest choice node that is not the first or second child of a loop node, i.e., $\times(a, b)$. Then apply transformation rule 1 of Definition 3 to obtain the tree in Fig. 6b. This tree contains two branches that are equal, except for the leaf nodes a and b. The probabilities of the original children of the choice, i.e., a and b, move up together with the choice operator.

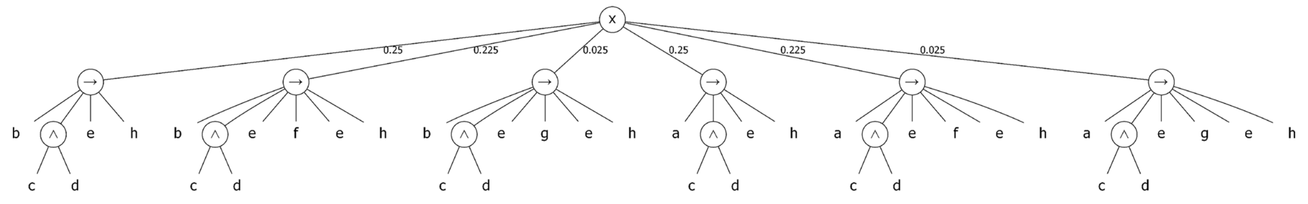
The two remaining choices under the root are both the second child of a loop node. To include these choices in LT dependencies, a loop unfolding step is needed. Consider for example an unfolding with a maximum of 1 repetition,[6] then the trace equivalent unfolded tree is shown in Fig. 6c.

Due to the unfolding of the loops, new choice nodes appear under the root node. Therefore, similarly to the first step, one can again apply transformation rule 1 to the choice $\times(f, g)$ (in the left and right branch) to obtain the tree in Fig. 6d. In the next step, apply transformation rule 2 to merge parent with child choices. This results in the tree in Fig. 6e. The probabilities of parent and child branches are multiplied when merging the choices.

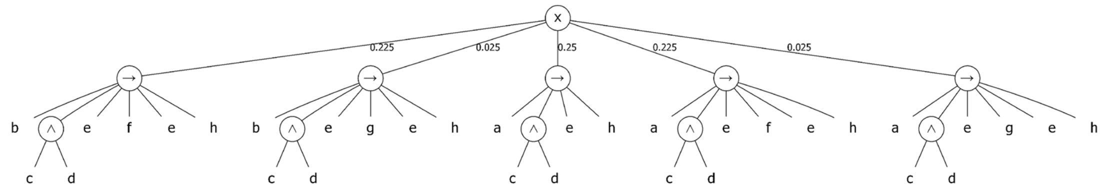
The unfolding of all choices continues until the root node of the tree is the only choice node in the tree (e.g., in Fig. 7a). If the user opts to not include the choices in the first or second child of a loop node in the dependencies, the unfolding stops when the root node is a choice and all other

[5] Tree $PT_1 = \circlearrowleft^k(\times(a, b), c, d)$ is not trace equivalent to tree $PT_2 = \times(\circlearrowleft^k(a, c, d), \circlearrowleft^k(b, c, d))$.

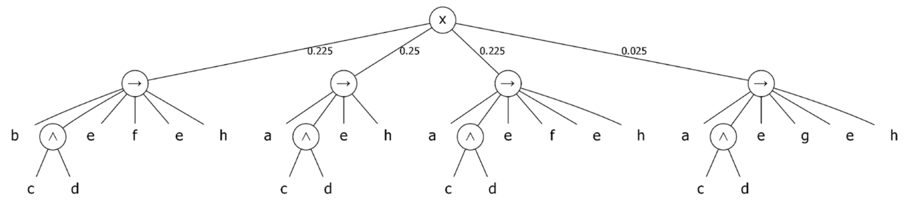
[6] Notice that activity e can be repeated once.



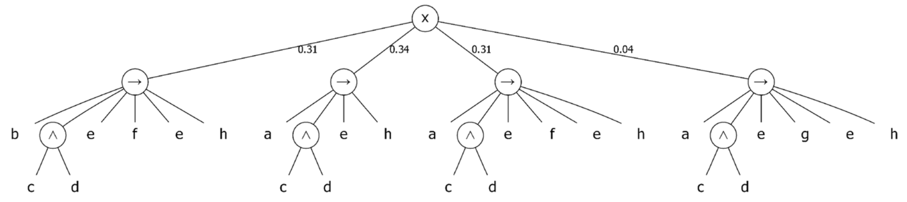
(a) Unfolded Choice Tree $PT_2^{\times}$



(b) After Removal of The First Branch



(c) After Removal of The Third Branch



(d) Normalized Branching Probabilities

Fig. 7 Inserting dependencies in unfolded choice tree $PT_2^{\times}$

choices are either a first or second child under a loop node (e.g., in Fig. 6b).

4.3.2 *Inserting Random Dependencies*

After the preparation step, the approach inserts random LT dependencies into the unfolded choice tree (see lines 19–24 of Algorithm 1). LT dependencies are created on a trace level while ensuring soundness. Each branch under the root node of the unfolded choice tree $PT^{\times}$ represents one combination of choice behavior in the original tree PT, i.e., a set of traces in the resulting event log. Removing a branch from the tree $PT^{\times}$ forbids this combination and thus inserts a LT dependency. To ensure random LT dependencies, the removal of a branch depends on the probability to insert LT dependencies Π^{Lt}.

To guarantee soundness, one could not simply remove any set of combinations of choice behavior, because some combinations together restrict too much choice behavior and thus result in dead activities. A dead activity occurs if an activity in the original tree PT does not occur in the tree $PT^{\times}$. The goal of the approach is to insert LT dependencies by limiting the choice behavior in a tree while preventing unsound behavior such as dead activities.

The pruning mechanism in Definition 4 prevents dead activities by checking if removing a branch from the root causes a dead activity in tree $PT^{\times}$. First, the mechanism retrieves all activities in the branch A_i. Then, it retrieves all activities in the other branches: A_o. If the activities in the selected branch are not contained in the set of activities of the other branches, i.e., $A_i \nsubseteq A_o$, then the selected branch cannot be removed.

Definition 4 (*Pruning Mechanism*) The pruning mechanism is a function $\phi : N \times N \rightarrow [\text{true}, \text{false}]$ that given the unfolded choice tree $PT^{\times}$ and a branch of the root, i.e., $c(r')_i$, returns 'false' if a dead activity occurs when eliminating the branch in the tree:

$$A_i = \{m(n) \in N_L | n \in s(c(r')_i)\}$$
$$A_o = \{m(n) \in N_L | n \notin s(c(r')_i)\}$$
$$\phi(PT^{\times}, c(r')_i) = \begin{cases} \text{true}, & \text{if } A_i \subseteq A_o. \\ \text{false}, & \text{if } A_i \nsubseteq A_o. \end{cases}$$

The insertion of LT dependencies is illustrated in Fig. 7. It starts from the unfolded choice tree $PT_2^{\times}$ (see Fig. 7a) obtained from unfolding PT_2 in Fig. 6. Then, Algorithm 1 visits each of the branches under the root choice node. Based on the probability to insert LT dependencies, $\Pi^{Lt} = 0.5$, the first, third and last branch are randomly selected as candidates for removal. The first and third branch can be removed as illustrated in Fig. 7b, c respectively. However, the pruning mechanism prevents removing the last branch as this would make 'g' a dead activity.

Finally, after removing the branches in $PT^{\times}$, the sum of the branching probabilities of the remaining children of the root does not equal to one: i.e., $\sum_{i=1}^{|c(r')|} b(c(r')_i) \neq 1$. Therefore, the branching probabilities of each of these child nodes are normalized (see lines 25–28 of Algorithm 1): for each node $n = c(r')_i$ with $i \in [1, |c(r')|]$ do $b(n) = b(n) / \sum_{i=1}^{|c(r')|} b(c(r')_i)$. In the example the branching probabilities of the tree in Fig. 7c are normalized as shown in Fig. 7d. This results in the final unfolded choice tree with LT dependencies which is sound:

Theorem 1 *Algorithm 1 generates unfolded choice trees with long-term dependencies that are sound.*

Proof The proposed algorithm generates LT dependencies on a trace level. It removes a set of branches from the unfolded choice tree to exclude some combinations of choice behavior in the tree. In this way choices are no longer free, but depend on other choices made earlier in the process, which conforms to the definition of *LT dependencies* (see introduction of Sect. 4.3).

The unfolded choice tree $PT^{\times}$ is created by applying the five transformation rules in Definition 3 and the loop unfolding step. The transformation rules use the operators $O = \{\rightarrow, \times, \wedge, \circlearrowleft^k, \vee\}$ and add duplicate activity labels, i.e., $m(n_1) = m(n_2) | n_1 \neq n_2$ and $n_1, n_2 \in N_L$, to unfold choices while ensuring trace equivalent behavior. The loop unfolding step replaces a loop operator ($\circlearrowleft^k$) by a combination of sequence and choice operators ($\rightarrow$, $\times$) plus a silent activity (τ) and duplicate activity labels. This loop unfolding ensures trace equivalence with the original bounded loop. As such the transformation rules plus the loop unfolding results in an unfolded choice tree that conforms with the Definition 1 of a block-structured process tree which is inherently block-structured and thus *sound*.

The removal of branches (subtrees) of the unfolded choice tree $PT^{\times}$ can never introduce deadlocks, but can produce dead activities by eliminating all branches in which a certain activity occurs. The pruning mechanism in Definition 4 prevents dead activities by ensuring at that each activity occurs in at least one branch of the final tree. The absence of dead activities together with the block-structuredness of $PT^{\times}$ guarantees *soundness*. □

Algorithm 1 :Insert random long-term dependencies

```
1: Input:
2:     $PT$: process tree
3:     $\Pi^{Lt}$: probability of inserting a LT dependency
4:     $UnfoldLoops$: whether or not to unfold loops
5:     $k$: maximum repetitions of loops
6: Output:
7:     $PT^{\times}$:unfolded choice tree with dependencies
8: Start InsertRandomLTDependencies($PT, \Pi^{Lt}$,
9:         $UnfoldLoops, k$)
10: $PT^{\times} \leftarrow PT$
11: while $\exists n \in s(r') | n \neq r'$ and $n \in N_{\times}$ do
12:     if UnfoldLoops = True then
13:         apply transformation rule to $n$ or unfold loop with maximum $k$ repetitions
14:     else
15:         apply transformation rule to $n$
16:     end if
17:     move the branching probabilities of $n$
18: end while
19: for $n \in c(r')$ do
20:     $x \leftarrow random$
21:     if $x < \Pi^{Lt}$ and $\phi(PT^{\times}, n)$ = true then
22:         remove entire branch $s(n)$ from $PT^{\times}$
23:     end if
24: end for
25: $z \leftarrow \sum_{i=1}^{|c(r')|} b(c(r')_i)$
26: for $n \in c(r')$ do
27:     $b(n) \leftarrow b(n)/z$
28: end for
29: return $PT^{\times}$
```

4.4 Sample Logs

This subsection focuses on the last step of the GED methodology: how to generate a sample of event logs from a sample of trees as generated in Sects. 4.2 and 4.3.

4.4.1 Setting Log Characteristics

The hierarchical design of the GED methodology (see Fig. 2) shows that one process tree represents a population of event logs. The population can be further refined using log characteristics. This paper uses two characteristics imposed by the full-control requirement in Sect. 2: the number of traces and the amount of noise. Similar to the model population, the user needs to specify each of these log characteristics.

Definition 5 formalizes a trace as a sequence of activities and an event log as a multiset of traces. The size of the log |*L*| is equal to the number of traces *t*. It expresses how many times the simulator will run from start to end through the process tree, logging each of these runs as a separate trace σ_j.

Definition 5 (*Trace, Event Log*) Let $A \subseteq \mathcal{A}$ be a finite set of activities. A trace $\sigma_j \in A^*$ is a sequence of activities. A log $L \in \mathbb{B}(A^*)$ is a multiset of traces. The size of the log is $|L| = t$.

This paper adopts the definition of noise by Günther (2009): "noise is incorrect behavior in the log that can be caused either by the logging mechanism or the constitution of the event data". The following types of noise behavior are adopted from Günther (2009): missing head, missing body (episode), missing tail, order perturbation and the introduction of additional activities. Assume a trace $\sigma_j = \langle a_1, \ldots, a_{n-1}, a_n \rangle$. The missing head, body and tail types, remove subsequences of a trace σ_j. The head of a trace contains activities a_i with $i \in [1, n/3]$, the body consists of activities a_i with $i \in [(n/3) + 1, 2n/3]$ and the tail contains activities a_i with $i \in [(2n/3) + 1, n]$. The order perturbation type interchanges two random activities. The additional activities type introduces a random activity in the trace.

The amount of noisy traces t^* in a log is specified using a binomial distribution: $t^* \sim Binomial(|L|, \Pi^{Noise})$. Π^{Noise} expresses the probability to select a trace for noise insertion. A noisy trace contains a random type of noise behavior which is decided based on a discrete uniform distribution. A trace with only one activity cannot be selected for noise insertion.

4.4.2 Simulating a Log From a Process Tree

This paper uses the principles of discrete-event simulation (DES) to simulate process trees with the SimPy simulation library (Matloff 2008). DES is a general and widely used simulation approach that models a process (system) as a series of events, i.e., instants in time when a state-change in the process occurs (Robinson 2014). The simulation of a DES model jumps in time from one event to the next in the series.

The simulation approach in this paper first translates a process tree into the general DES model components: process, activities, events and entities (Shannon 1977). In a next step, it simulates the obtained DES model into an event log. The detailed algorithms for translating a process tree into a DES model and simulating such a model are outside the scope of this paper, but the interested reader is referred to the technical paper (Jouck and Depaire 2017). The implementations of the algorithms are based on the SimPy simulation library (Matloff 2008)and are available on Github: https://github.com/tjouck/PTandLogGenerator..

After the simulation, the noise insertion step will occur. This step iterates over all the traces in the log. Based on the probability Π^{Noise} a trace is selected. The selected trace randomly gets one of the noise types discussed above. Finally, the resulting log serves as input for process discovery evaluation.

5 Demonstration and Evaluation

The previous sections focused on the design and development of the GED methodology and its implementation. This section will discuss the next steps within the design science framework: the demonstration and evaluation of the GED methodology.

5.1 Tool Implementation

Empirical analysis of CF Discovery algorithms typically requires an extensive set of experiments. Therefore, the implementation of the GED methodology should be automated for its application in empirical analysis. At the same time, the automation needs to comply to the third group of requirements of the GED methodology, i.e. standard formats and integration within the ProM Framework (Verbeek et al. 2011). For the standard formats, this means that the output process trees and event logs should be in the PTML and XES standard formats (Verbeek et al. 2011) respectively.

Two tool implementations are available: one Python package and one package with plugins in the ProM framework. The Python package is available on Github.[7] The package contains programs callable from command line for generating random process trees and generating event logs from those trees. The ProM package *PTandLogGenerator* (Jouck and Depaire 2016) includes the plugins 'Generate Process Trees from Population' and 'Generate Log Collection (with noise) from Process Trees'. Each of the tools support the necessary standard formats.

5.2 Data Generation Setup

To demonstrate the GED methodology and its implementation, a use case is designed. The use case evaluates the performances of a set of CF discovery algorithms. In such a case, the evaluation requires multiple process models and

[7] https://github.com/tjouck/PTandLogGenerator.

Table 3 Input parameters of data generation

Parameter	Population **MP**$_{\textbf{new}}$	Population **MP**$_{\textbf{existing}}$	Population **MP**$_{\textbf{scalability}}$
Number of visible activities	(10,20,30)	(10,20,30)	(10,20,30)
Sequence ($\Pi^{\rightarrow}$)	0.5	0.5263158	$\in$[0,1]
Parallel ($\Pi^{\wedge}$)	0.15	0.1578947	$\in$[0,1]
Choice ($\Pi^{\times}$)	0.25	0.2631579	$\in$[0,1]
Loop ($\Pi^{\circlearrowleft}$)	0.05	0.0526316	$\in$[0,1]
Or ($\Pi^{\vee}$)	0.05	0.0	$\in$[0,1]
Silent activities (Π^{τ})	0.1	0.1	0.1
Reoccurring activities (Π^{Re})	0.1	0.0	0.1
Long-term dependencies (Π^{Lt})	0.5	0.0	$\in$[0,1]
Unfold loops	True	/	$\in${False,True}
Max repeat (k)	1	/	$\in${0,1,2}
Infrequent paths (Π^{In})	0.5	0.5	0.5
Sample size (number of trees)	2000	50	1000
Logs per model	1	1	/
Number of traces (t)	1000	1000	/
Noise (Π^{Noise})	0.1	0.1	/

event logs with an extensive set of control-flow patterns to avoid an oversimplified evaluation.

In the first step, the model populations are defined (see Table 3). The definition of a model population *MP* (see Definition 2) requires the specification of the top 12 parameters in column 1 of Table 3. This demonstration uses two model populations $\mathrm{MP}_{\mathrm{New}}$ and $\mathrm{MP}_{\mathrm{Existing}}$ each with different parameter settings, except for the number of visible activities which varies between 10 and 30. The $\mathrm{MP}_{\mathrm{New}}$ population contains all the base patterns, silent and reoccurring activities and choices with infrequent paths. Additionally it contains LT dependencies for which loops with choices are unfolded with maximum one repetition. The $\mathrm{MP}_{\mathrm{Existing}}$ population contains all the patterns available in current state-of-the-art tool PLG2 (Burattin 2015). Therefore, $\mathrm{MP}_{\mathrm{Existing}}$ does not contain 'or', reoccurring activities and LT dependencies.

In the second step, a sample of models is drawn from each model population (see Table 3). Finally, in the third step, the simulator will generate event logs from the trees in the sample. The simulation parameters to set are the number of logs per tree, the number of traces in the log and the probability of noise insertion (see Table 3). For the two model populations, the demonstration will generate one event log per tree, each log containing 1000 traces and 10% noise probability.

The setup of the parameters as in Table 3 can serve as a template for future users of the GED methodology in empirical CF discovery analysis. Including this table in the report of such an analysis will clearly describe the event data used in the experiments and also enhance transparency and reproducibility of the experiment results.

5.3 Evaluation

The evaluation investigates whether the GED methodology implementation meets all the requirements stated in Table 1. Additionally, it will assess the scalability of the implementation. Finally, an empirical evaluation of four process discovery techniques validates the effectiveness of the GED methodology.

5.3.1 Requirements

The *full control* requirements imply that a user can control the control-flow behavior in the generated process trees (control-flow patterns) and event logs (log characteristics). Therefore, this part of the evaluation checks whether the characteristics of the sample of trees and logs of the use case conform with the input parameters of population $\mathrm{MP}_{\mathrm{New}}$ in Table 3. Table 4 displays the descriptive statistics of the tree and log sample characteristics drawn from population 1.

Firstly, the distribution of the number of visible activities conforms the triangular distribution characterized in Table 3. Secondly, the mean relative frequencies and the confidence intervals for these means of all the control-flow constructs are shown in the second and third column of

Table 4 Descriptive statistics of a sample from population MP_{New}

Parameter	Sample mean	Confidence interval	Population value in CI?
Number of visible activities	(11,21,30)	/	/
Sequence ($\Pi^{\rightarrow}$)	0.4982	[0.4931, 0.5032]	True
Parallel ($\Pi^{\wedge}$)	0.1506	[0.1470, 0.1542]	True
Choice ($\Pi^{\times}$)	0.2544	[0.2502, 0.2586]	False
Loop ($\Pi^{\circlearrowleft}$)	0.0471	[0.0449, 0.0493]	False
Or ($\Pi^{\vee}$)	0.0498	[0.0475, 0.0520]	True
Silent activities (Π^{τ})	0.0976	[0.0921, 0.1032]	True
Reoccurring activities (Π^{Re})	0.0987	[0.0958, 0.1016]	True
Long-term dependencies (Π^{Lt})	0.3835	[0.3753, 0.3917]	False
Infrequent paths (Π^{In})	0.4833	[0.4708, 0.4958]	False
Tree generation time (seconds)	17.849	[2.9137, 32.784]	/
Mean percentage of noisy traces	0.0934	[0.0924, 0.0943]	False
Log generation time (seconds)	1.37	[1.3421, 1.3894]	/

Table 4.[8] The population values of most parameters are contained in the confidence interval of the mean and some only differ slightly from the interval. A noticeable exception is the confidence interval for LT dependencies, which is more than 10% points lower than the population value. This was caused by the pruning mechanism which prevents inserting LT dependencies that cause dead activities. As such the average percentage of LT dependencies a tree will mostly be below the population value.

The number of traces in the generated event logs are exactly as specified in the input parameters. The average percentage of noisy traces is slightly lower than the probability set in Table 3. This percentage was influenced by not considering traces with only one activity which has led to fewer than 1000 candidate traces in some logs.

Overall, the implementation satisfies the full control requirement as it effectively allows users to control the characteristics of the generated models and logs through a population. Note that the soundness requirement was already proven by Theorem 1 in Sect. 4.3.

Next, to the input parameters, Table 4 shows the mean tree and log generation time in seconds. These performances were accomplished on a laptop with an Intel Core i5-4200U processor and 8 GB of RAM memory.

The *randomness* requirement implies that the generation of the trees and logs should be done in a random way. Both the ProM and Python tool implementations support such a random generation. The subsection describing the tool implementation already mentioned that both tools meet the *formats* requirement.

5.3.2 *Scalability*

This subsection describes an analysis done in order to assess the scalability of the tree generation.[9] This part of the evaluation studies the relation between tree generation time and model population parameters. The unfolding of a tree into the unfolded choice tree is the most expensive operation in terms of computation time. Such unfoldings happen when choice and loop constructs appear in the tree and the probability to insert LT dependencies is larger than 0 (see Sect. 4.3). Therefore, 1000 model populations are specified with varying probabilities and settings ($MP_{scalability}$ in column 4 in Table 3):

- The probabilities of ‘sequence’, ‘choice’, ‘parallel’, ‘or’ and ‘loop’ vary between 0 and 1 while ensuring the sum is equal to 1.
- The probability of LT dependencies varies between 0 and 1.
- The unfolding of loops with choices in the first or second child has a probability equal to 50%
- If loops are unfolded, then the maximum number of repititions of the loop varies between 0 and 2

From each of the 1000 model populations, one random tree is generated. The tree generation aborts after 10,000 s. In total 23 trees, i.e. 2.3% of all generated trees, were aborted. The other 977 trees have a median generation time of 0.63 s and a minimum and maximum of respectively 0.03 and 8736 s. To understand which model parameters influence the long tree generation, spearman correlation coefficients were calculated. Table 5 shows that there are only 4 small, yet significant positive correlation coefficients using a 5% significance level. When the probability of a loop construct

[8] These mean relative frequencies of the operator types were calculated before the trees were reduced.

[9] The scalability of the log generation is outside the scope of this paper as the focus is on model generation with LT dependencies.

Table 5 Positive significant correlations between tree generation time (‘Time’) or aborting tree generation (‘Aborted’) and model population parameters

Variable 1	Variable 2	Spearman correlation	P-value
Loop ($\Pi^{\circlearrowleft}$)	Aborted	0.12	1.19e−04
Max repeat	Aborted	0.19	1.43e−09
Choice ($\Pi^{\times}$)	Time	0.32	1.39e−24
Max repeat	Time	0.19	7.73e−40

or the maximum number of loop repetitions increases, then the probability of exceeding 10,000 s for tree generation tends to increase. Similarly, when the probability of a choice construct or the maximum number of loop repetitions increases, the tree generation time tends to increase. Overall, it is hard to predict a long tree generation time using only model parameters. One could assign more computing time or use statistical techniques that can handle missing values, e.g., truncated data analysis, to handle the exceptionally long tree generation times.

In comparison, trees without LT dependencies never suffer from exceptionally long generation times. An additional experiment specified another 1000 model populations without LT dependencies and varying probabilities of ‘sequence’, ‘choice’, ‘parallel’, ‘or’ and ‘loop’ as before. Again one tree is generated from every model population. Each of those trees could be generated within 2 s. All performances were accomplished on a laptop with an Intel Core i5-4200U processor and 8 GB of RAM memory.

5.3.3 *Effectiveness*

The final part of the evaluation asserts the effectiveness of the GED methodology and implementation. It tests the hypothesis that an evaluation with the GED methodology and implementation leads to new insights that could not be obtained by using the current state-of-the art technique PLG2 (Burattin 2015). For this purpose, an empirical evaluation with four discovery algorithms, α_{++} (Wen et al. 2007), ILP (van derWerf et al. 2009), Inductive (Leemans et al. 2014) and Flexible Heuristics (Weijters and Ribeiro 2011), on two model populations has been done. The first model population ($MP_{existing}$) contains models with all constructs supported by PLG2, the second model population (MP_{new}) additionally contains the constructs ‘or’, reoccurring activities and LT dependencies as supported by GED methodology and implementation. Columns two and three of Table 3 display the specific parameter settings for each of the constructs. Notice that the proportions between the constructs sequence, choice, parallel and loop constructs is kept constant, e.g., $\Pi^{\times}/\Pi^{\rightarrow} = 0.5$. If the above stated hypothesis is true, discovery algorithms will perform differently on the two populations.

The evaluation first draws a random sample of 50 models from each population. Then, one log per model is simulated containing 1000 traces and 10% of noise using a combination of the noise operators in Sect. 4.4. Then, all four discovery algorithms mine a model from each log and the quality of each discovered model with regard to that log is measured in terms of fitness and precision using the alignment based fitness and precision metrics (Van der Aalst et al. 2012). These two metrics are combined in one value using the F1-score, i.e., the harmonic mean of fitness and precision: $\frac{2 \cdot precision \cdot fitness}{precision + fitness}$. Finally, the differences in fitness, precision and F1 values of the process discovery algorithms are compared statistically.

Table 6 shows an overview of the obtained results: column two contains the results for $MP_{existing}$, while column three shows the results for MP_{new}. For each quality dimension the average rank for each discovery algorithm is shown. The algorithms are sorted with the best performing algorithm (with the highest rank) on top. The Friedman test (Demsar 2006) is applied to determine whether there is a significant difference in performance of the discovery technique. The results indicate that the techniques do *not* perform equivalently for each combination of quality dimension and dataset, i.e., the null hypothesis is rejected using a 95% confidence interval. This is followed by a Wilcoxon signed rank test (Benavoli et al. 2016; Demsar 2006) to test the significance of each of the pairwise differences between algorithms using a Bonferroni corrected significance level to guarantee that the family-wise Type I

Table 6 Average rankings for process discovery algorithms for each quality dimension within a model population

Quality metric	$\mathbf{MP_{existing}}$	$\mathbf{MP_{new}}$
Fitness	ILP (4.0)	ILP (4.0)
	$\underline{\text{Heuristics (2.54)}}$	$\underline{\text{Inductive (2.52)}}$
	$\underline{\text{Inductive (2.46)}}$	$\underline{\text{Heuristics (2.42)}}$
	α_{++} (1.0)	α_{++} (1.06)
Precision	Heuristics (3.6)	$\underline{\text{Heuristics (3.36)}}$
	Inductive (3.18)	$\underline{\text{Inductive (2.94)}}$
	$\underline{\text{ILP (1.62)}}$	α_{++} (2.28)
	$\underline{\alpha_{++}\text{(1.6)}}$	ILP (1.42)
F1	Heuristics (3.68)	$\underline{\text{Heuristics (3.5)}}$
	Inductive (3.28)	$\underline{\text{Inductive (3.2)}}$
	ILP (1.78)	ILP (1.9)
	α_{++} (1.26)	α_{++} (1.4)

Pairs of techniques that do not differ statistically from each other at the 95% confidence level are underlined error is smaller than 5%. Pairs of techniques that do not differ statistically are underlined.

In the fitness dimension the order between Heuristics and Inductive miner is different for the two datasets. However, the difference between these two miners is not statistically significant. In the precision dimension the order between ILP and α_{++} miner is different for the two datasets, yet the difference between the algorithms is not statistically significant for the $\mathrm{MP}_{\mathrm{existing}}$ dataset. Also in the precision dimension, the difference between Heuristics and Inductive miner is only statistically significant for the $\mathrm{MP}_{\mathrm{existing}}$ dataset. Looking beyond the average rankings, the Heuristics miner outperforms Inductive 36 times for the $\mathrm{MP}_{\mathrm{existing}}$ dataset while it decreases to 33 times for the $\mathrm{MP}_{\mathrm{new}}$ dataset. Finally, all differences in terms of F1 between miners are statistically significant for the $\mathrm{MP}_{\mathrm{existing}}$ dataset, while for the $\mathrm{MP}_{\mathrm{new}}$ dataset the difference between Heuristics and Inductive miner is not statistically significant.

Overall, the conclusion of the analysis is that the difference between Heuristics and Inductive miner becomes smaller in terms of precision for models with 'or', reoccurring activities and LT dependencies. Conversely, the difference between α_{++} and ILP in terms of precision becomes larger for such models. These observations show that the extra constructs have negative effects on the Heuristics, Inductive and ILP miner (only on precision), while it has positive effects on the α_{++} miner. Moreover, the negative effects on Heuristics miner are larger than the negative effects on Inductive miner. As such, these observations provide evidence for the hypothesis that evaluation with the GED methodology and implementation leads to new insights that could not be obtained by using the current state-of-the art technique PLG2 (Burattin 2015). This demonstrates the effectiveness of the proposed methodology and implementation.

6 Conclusions

This paper introduces the GED methodology and implementation to generate artificial event data for empirical CF discovery analysis. It involves three steps: defining a model population, drawing a sample of models from that population and simulating the sample of models into a sample of event logs. The demonstration and evaluation show that the implementation of the GED methodology succeeds in generating artificial data for empirical process discovery analysis such that:

- the generated models are random samples of predefined populations, allowing for a wide range of suitable (confirmatory) statistical experimental analysis,
- the populations allow for more complex process models than the existing approaches do (including LT dependencies, 'or' and duplicate activities),
- the approach is sufficiently performant for large scale experiments,
- the approach is able to reveal insights which remained hidden when considering simpler process populations (which were the only ones the existing techniques could handle so far).

The implementation in this paper does not claim to have achieved full control over all possible control-flow patterns. However, it includes all patterns that were frequently used in CF discovery comparisons. Moreover, the definition of LT dependencies in this paper focuses on dependencies between exclusive choices, but, in future work this definition could be extended to allow for dependencies between non-exclusive choices.

Due to the focus on CF discovery, the scope of the methodology and implementation is limited to the control-flow perspective. However, new process discovery algorithms also focus on extracting knowledge in other perspectives such as time, resources and data (e.g., de Leoni et al. 2016). Naturally, the empirical analysis of such techniques requires event data that include these other perspectives as well. As a result, new challenges and opportunities for extending the proposed GED methodology and implementation arise, for example, the inclusion of LT dependencies based on the data attributes of the process instances.

Acknowledgements The authors would like to thank Massimiliano de Leoni and Alfredo Bolt for their advice and support to implement the PTandLogGenerator.

References

Benavoli A, Corani G, Mangili F (2016) Should we really use post-hoc tests based on mean-ranks? J Mach Learn Res 17:1–10. http://jmlr.org/papers/v17/benavoli16a.html. Accessed 21 Oct 2017

Box GE, Hunter JS, Hunter WG (2005) Statistics for experimenters: design, innovation, and discovery, vol 2. Wiley, New York. http://stats.cwslive.wiley.com/details/book/2791421/Statistics-for-Experimenters-Design-Innovation-and-Discovery-2nd-Edition.html. Accessed 16 Jan 2017

Buijs JCAM (2014) Flexible evolutionary algorithms for mining structured process models. PhD thesis, Technische Universiteit Eindhoven, Eindhoven. http://alexandria.tue.nl/extra2/780920.pdf. Accessed 23 Feb 2015

Burattin A (2015) PLG2: Multiperspective processes randomization and simulation for online and offline settings. Tech. rep., University of Innsbruck. https://arxiv.org/abs/1506.08415. Accessed 28 July 2016

Burattin A, Sperduti A (2011) PLG: a framework for the generation of business process models and their execution logs. In: business process management workshops, Springer, Heidelberg, pp 214–219. http://link.springer.com/chapter/10.1007/978-3-642-20511-8sps20. Accessed 06 Jan 2015

De Weerdt J, De Backer M, Vanthienen J, Baesens B (2012) A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Inf Syst 37(7):654–676. https://doi.org/10.1016/j.is.2012.02.004. http://www.sciencedirect.com/science/article/pii/S0306437912000464. Accessed 10 Dec 2013

de Leoni M, van der Aalst WM, Dees M (2016) A general process mining framework for correlating, predicting and clustering dynamic behavior based on event logs. Inf Syst 56:235–257. http://www.sciencedirect.com/science/article/pii/S0306437915001313. Accessed 19 Dec 2016

de Medeiros AKA, Weijters AJ, van der Aalst WM (2007) Genetic process mining: an experimental evaluation. Data Min Knowl Discov 14(2):245–304. http://link.springer.com/article/10.1007/s10618-006-0061-7. Accessed 26 May 2014

Demsar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30. http://dl.acm.org/citation.cfm?id=1248548. Accessed 26 May 2014

Dumas M, La Rosa M, Mendling J, Reijers HA (2013) Fundamentals of business process management. Springer, Heidelberg. http://link.springer.com/content/pdf/10.1007/978-3-642-33143-5.pdf. Accessed 15 Sept 2015

Günther CW (2009) Process mining in flexible environments. PhD thesis, Technische Universiteit Eindhoven. http://www.narcis.nl/publication/RecordID/oai:library.tue.nl:644335. Accessed 04 Apr 2016

Jensen K, Kristensen LM, Wells L (2007) Coloured petri nets and CPN tools for modelling and validation of concurrent systems. Int J Softw Tools Technol Transf 9(3-4):213–254. http://link.springer.com/article/10.1007/s10009-007-0038-x. Accessed 26 May 2014

Jin T, Wang J, Wen L (2011) Efficiently querying business process models with BeehiveZ. In: BPM (Demos), http://ceur-ws.org/Vol-820/Demo1.pdf. Accessed 06 Nov 2013

Johannesson P, Perjons E (2014) An introduction to design science. Springer, Heidelberg. http://books.google.be/books?hl=nl&lr=&id=ovvFBAAAQBAJ&oi=fnd&pg=PR5&dq=an+introduction+to+design+science&ots=r45U7mRMl4&sig=FuiaKJ1PoRCdgTev-IQAIebgcfE. Accessed 05 Dec 2014

Jouck T, Depaire B (2016) PTandLogGenerator: a generator for artificial event data. In: Proceedings of the BPM Demo Track 2016 (BPMD 2016), CEUR workshop proceedings, Rio de Janeiro, vol 1789, pp 23–27. http://ceur-ws.org/Vol-1789/. Accessed 05 Jan 2018

Jouck T, Depaire B (2017) Simulating process trees using discrete-event simulation. Technical Report, Hasselt University. https://uhdspace.uhasselt.be/dspace/handle/1942/23130. Accessed 05 Jan 2018

Kataeva V, Moscow RF, Kalenkova AA (2014) Applying graph grammars for the generation of process models and their logs. In: Proceedings of the spring/summer young researchers colloquium on software engineering, http://syrcose.ispras.ru/2014/files/submissions/12spssyrcose2014.pdf. Accessed 19 Dec 2014

Leemans SJ, Fahland D, van der Aalst WM (2014) Discovering block-structured process models from event logs containing infrequent behaviour. In: Business process management workshops, Springer, Heidelberg, pp 66–78. http://link.springer.com/chapter/10.1007/978-3-319-06257-0sps6. Accessed 27 Oct 2015

Matloff N (2008) Introduction to discrete-event simulation and the simpy language. Davis, CA Dept of Computer Science University of California at Davis Retrieved on August 2:2009. http://web.cs.ucdavis.edu/~matloff/matloff/publicspshtml/SimCourse/PLN/DESimIntro.pdf. Accessed 20 Mar 2016

Mitsyuk AA, Shugurov IS, Kalenkova AA, van der Aalst WM (2017) Generating event logs for high-level process models. Simul Model Pract Theor 74:1–16. http://www.sciencedirect.com/science/article/pii/S1569190X17300047. Accessed 05 Jan 2018

Robinson S (2014) Simulation: the practice of model development and use. Palgrave Macmillan, Basingstoke. https://books.google.be/books?hl=nl&lr=&id=TEMdBQAAQBAJ&oi=fnd&pg=PP1&dq=Simulation+%E2%80%93+The+practice+of+model+development+and+use&ots=XIP9NsOH2J&sig=ASAxxwYB2hSCFVqaAAuJFe4nBbs. Accessed 25 Aug 2016

Rozinat A, de Medeiros AA, Gnther CW, Weijters A, van der Aalst WM (2007) Towards an evaluation framework for process mining algorithms. BPM Center Report BPM-07-06, http://alexandria.tue.nl/repository/books/630086.pdf. Accessed 04 Feb 2014

Russell N, ter Hofstede AHM, van der Aalst WMP, Mulyar N (2006) Workflow controlflow patterns: a revised view. Tech. Rep. 06-22. https://www.bpmcenter.org/. Accessed 20 Feb 2015

Shannon RE (1977) Introduction to simulation languages. In: Proceedings of the 9th conference on winter simulation-Volume 1, winter simulation conference, pp 14–20. http://dl.acm.org/citation.cfm?id=807515. Accessed 22 Nov 2016

Stocker T, Accorsi R (2013) Secsy: security-aware synthesis of process event logs. In: Workshop on enterprise modelling and information systems architectures, pp 71–84. https://pdfs.semanticscholar.org/fa29/18da96fa73fe6430233ea6b9403c86fd6797.pdf. Accessed 05 Jan 2018

Van der Aalst W, Adriansyah A, van Dongen B (2012) Replaying history on process models for conformance checking and performance analysis. Wiley Interdiscip Rev Data Min Knowl Discov 2(2):182–192. http://onlinelibrary.wiley.com/doi/10.1002/widm.1045/full. Accessed 06 Feb 2015

van derWerf JME, van Dongen BF, Hurkens CA, Serebrenik A (2009) Process discovery using integer linear programming. Fund Inf 94(3):387–412. http://iospress.metapress.com/index/CX85102T26280611.pdf. Accessed 26 May 2014

van Dongen BF, De Medeiros AA, Wen L (2009) Process mining: overview and outlook of petri net discovery algorithms. In: Transactions on petri nets and other models of concurrency II, Springer, Heidelberg, pp 225–242. http://link.springer.com/chapter/10.1007/978-3-642-00899-3sps13. Accessed 01 June 2014

van Hee KM, Liu Z (2010) Generating benchmarks by random stepwise refinement of petri nets. In: ACSD/Petri Nets Workshops, pp 403–417. http://ceur-ws.org/Vol-827/31spsKeesHeespsarticle.pdf?origin=publicationspsdetail. Accessed 25 Nov 2014

van der Aalst W (2016) Process mining: data science in action. Springer, Heidelberg. https://books.google.be/books?hl=nl&lr=&id=hUEGDAAAQBAJ&oi=fnd&pg=PR5&dq=process+mining+data+science+in+action&ots=ZBhPEo-BpL&sig=Ahy9qBgJGES4kWX3NnsNeGu6ekY. Accessed 17 Nov 2016

van der Aalst WMP (1998) The application of Petri nets to workflow management. J Circ Syst Comput 8(01):21–66. http://www.worldscientific.com/doi/abs/10.1142/S0218126698000043. Accessed 20 Feb 2015

van der Aalst W, Buijs J, Van Dongen B (2012) Towards improving the representational bias of process mining. In: Data-driven process discovery and analysis, Springer, Heidelberg, pp 39–54. http://link.springer.com/chapter/10.1007/978-3-642-34044-4_3. Accessed 20 Feb 2015

vanden Broucke SK, Delvaux C, Freitas J, Rogova T, Vanthienen J, Baesens B (2014) Uncovering the relationship between event log characteristics and process discovery techniques. In: Business process management workshops, Springer, Heidelberg, pp 41–53. http://link.springer.com/chapter/10.1007/978-3-319-06257-0sps4. Accessed 19 Sept 2014

Verbeek HMW, Buijs JCAM, Van Dongen BF, Van Der Aalst WMP (2011) Xes, xesame, and prom 6. In: Soffer P, Proper E (eds) Information systems evolution, Springer, Heidelberg, pp 60–75. http://link.springer.com/chapter/10.1007/978-3-642-17722-4sps5. Accessed 20 Feb 2015

Wang J, Wong RK, Ding J, Guo Q, Wen L (2012) On recommendation of process mining algorithms. In: IEEE 19th International conference on web services (ICWS), IEEE, pp 311–318. http://ieeexplore.ieee.org/xpls/absspsall.jsp?arnumber=6257822. Accessed 30 Dec 2015

Weber P, Bordbar B, Tino P (2013) A framework for the analysis of process mining algorithms. IEEE Transact Syst Man Cybern Syst 43(2):303–317. http://ieeexplore.ieee.org/xpls/absspsall.jsp?arnumber=6202711. Accessed 16 Jan 2017

Weijters A, Ribeiro J (2011) Flexible heuristics miner (FHM). In: 2011 IEEE symposium on computational intelligence and data mining (CIDM), pp 310–317

Wen L, van der Aalst WM, Wang J, Sun J (2007) Mining process models with non-free-choice constructs. Data Min Knowl Discov 15(2):145–180. http://link.springer.com/article/10.1007/s10618-007-0065-y. Accessed 26 May 2014