Made available by Hasselt University Library in https://documentserver.uhasselt.be

Simulating Process Trees Using Discrete-Event Simulation

Peer-reviewed author version

JOUCK, Toon & DEPAIRE, Benoit (2017) Simulating Process Trees Using Discrete-Event Simulation.

DOI: 10.13140/RG.2.2.25488.00009
Handle: http://hdl.handle.net/1942/23130

Simulating Process Trees Using Discrete-Event Simulation

Toon Jouck and Benoît Depaire

Hasselt University, Faculty of Business Economics,
Agoralaan Bldg D, 3590 Diepenbeek, Belgium
toon.jouck@uhasselt.be; benoit.depaire@uhasselt.be

Abstract. The Process Tree notation is an emerging language for modeling block-structured processes. A Process Tree is inherently sound and therefore proves to be the ideal input of a simulator as it can never deadlock. However, most business process simulation tools require a translation to Petri Nets. This technical paper proposes a simulation tool for Process Trees based on the principles of discrete-event simulation (DES) that accepts a Process Tree as input. The proposed implementation benefits from the extensive work that has already been done on DES, including software packages to execute DES simulation. The simulation algorithms and implementations are free to download and use.

Keywords: artificial event logs; process trees; simulation

1 Introduction

Process mining techniques can be used to extract process-related information based on event data stored in event logs [1]. In order to test and improve process mining techniques, researchers need event logs [2]. One way to obtain event logs is to create synthetic event logs by simulating process models. In contrast to real event logs, the user has full control over the characteristics of the synthetic event logs. As a result, a synthetic log is tailor-made for testing specific aspects of a process mining technique.

Many tools exist for simulating business process models into event logs, e.g. [3,4,5,2]. These implementations work with respectively Coloured Petri Nets, BPMN, Petri Nets and Declare Models as language of the input models. However, none of the existing tools accepts a Process Tree as input. Process Trees are an emerging language for modeling block-structured processes [6,7,8]. A Process Tree is inherently sound and as a result is an ideal input for a simulator as it can never deadlock.

The main contribution of this paper is a new simulation tool based on the principles of discrete-event simulation (DES) that accepts a Process Tree as input. The user can set the number of process instances and the amount of noise in the final event logs. The format of the generated event logs conforms to the XES standard [9]. The proposed tool focuses on simulating the control-flow perspective of processes. As such, the generated event logs serve as input of control-flow discovery techniques.

The paper starts with a section 2 on Process Trees, event logs and noise. In the next section 3 the simulation algorithms are described. Section 4 discusses the tool implementation. Finally, section 5 concludes the paper and discusses future work.

2 Preliminaries

2.1 Process Trees

Definition 1 formalizes the Process Tree notation used in this paper. The definition is adopted from [7] and extended with the parent (p) and probability mapping (b) functions.

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*
- *$O = \{\rightarrow, \times, \wedge, \circlearrowleft, \vee\}$ 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:*
    $\forall n \in N : \forall_{1 \leqslant i < j \leqslant |c(n)|} : c(n)_i \neq c(n)_j$;
  - *a node with a loop operator type has exactly three children:*
    $\forall n \in N : (m(n) = \circlearrowleft) \Rightarrow |c(n)| = 3$.
- *$p : N \rightarrow N$ is the parent relation function:*
  $p(n) = k \Leftrightarrow n \in c(k)$
- *each node has a probability of being chosen:*
  $b : N \rightarrow [0, 1]$ is a mapping function mapping a probability to each node n:

$$b(n) = \begin{cases} 1, & \text{if } p(n) \notin N_\times \\ \in [0, 1], & \text{such that } \sum_{k}^{k \in c(p(n))} b(k) = 1 \text{ if } p(n) \in N_\times. \end{cases}$$

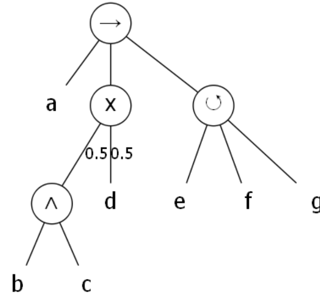
- *Let N^* be the set of all finite sequences over N then $s : N \to N^*$ is the subtree function, returning all nodes of n in a pre-order:*

$$s(n) = \begin{cases} n, & \text{if } n \in N_L. \\ n \cdot s(c(n)_1) \cdot \ldots \cdot s(c(n)_{|c(n)|}), & \text{if } n \in N_O. \end{cases}$$

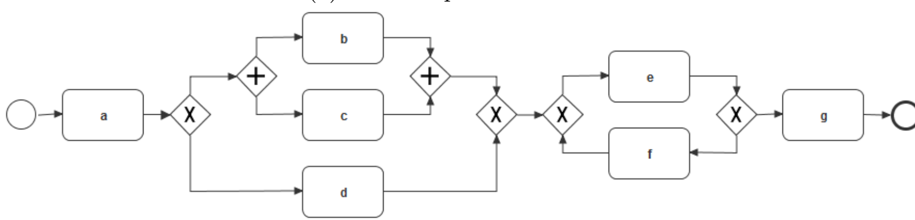
- *A node $n \in N$ can be denoted in shorthand as follows: $n = t\langle n_1, \ldots, n_k \rangle$ where $t = m(n)$ and $\langle n_1, \ldots, n_k \rangle = c(n)$.*

The process tree operator semantics in this paper are adopted from [7]. There is a trace equivalent Petri Net (or BPMN) translation for each operator (see page 59 and 63 in [7]). Leaf nodes with a τ label represent invisible activities.

Figure 1a shows an example Process Tree with its trace equivalent BPMN representation. The root node of the tree is a sequence node and therefore all of its children will be executed from left to right. The process starts with activity 'a' followed by a choice that executes both 'b' and 'c' in parallel or activity 'd'. The choice node is balanced as the probabilities of the branches under the choice are both 50%, i.e. each child branch is executed in 50% of the cases. After the choice, there is a loop node that starts with activity 'e', which can be followed a finite number of times by activity 'f' and then executing activity 'e' again. The process ends with the execution of activity 'g'.



(a) An example Process Tree



(b) Example Process Tree as BPMN model

Fig. 1: Example Process Tree and equivalent BPMN model

2.2 Event logs

Definition 2 formalizes a trace as a sequence of activities and an event log as a multiset of traces.

Definition 2 (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 $|L| = t$.*

De Medeiros [10] defines noise as low-frequent incorrect behavior in the log. The following types of noise behavior are adopted from [11]: missing head, missing body, missing tail, swap tasks and remove task. Definition 3 describes each of these noise types. None of the listed noise types can be applied to a trace with only one activity.

Definition 3 (Noise Types). *Assume a trace $\sigma_j = \langle a_1, \ldots, a_{n-1}, a_n \rangle$ then:*

- *Missing head: removes all activities a_i in σ_j with $i \in [1, \frac{n}{3}]$*
- *Missing body: removes all activities a_i in σ_j with $i \in [(\frac{n}{3}) + 1, \frac{2n}{3}]$*
- *Missing tail: removes all activities a_i in σ_j with $i \in [(\frac{2n}{3}) + 1, n]$*
- *Swap tasks: interchanges two random activities a_i and a_j in σ_j with $i \neq j$*
- *Remove task: remove random activity a_i in σ_j*

3 Simulation Approach

3.1 Input Parameters

A process model is the main input of the simulator and can be seen as an event log population. Therefore, an event log represents a random observation from that population. The user can influence which observations can be drawn from the population by setting simulation parameters. The presented simulation approach focuses on the control-flow perspective. Therefore, the user can set the parameters size and noise.

The size of the log $|L|$ is equal to the number of traces t it contains. The size equals the number of times the simulator will run from start to end through the Process Tree, logging each of these runs as a separate trace σ_j. The amount of noise in a log is another simulation parameter. The number of noisy traces follows a binomial distribution, i.e. $t' \sim Binomial(|L|, \Pi^{Noise})$ with $t' \leqslant t$. Π^{Noise} represents the probability to select a trace for noise insertion. A selected trace will get a random type of noise behavior (see Definition 3) based on a uniform distribution, i.e. each type has an equal probability. Traces with only one activity are omitted from noise insertion.

3.2 Simulation Algorithm

This section introduces the algorithms to simulate Process Trees using DES. The algorithm 1 describes all the steps in the approach, which are also visualized in Fig. 2. DES models a process, here a Process Tree, as as series of events. These events correspond to instants in time when a state-change in the process occurs [12]. The simulator of the DES model jumps from one event to the next in the series. Each of these events are logged to obtain an event log.

Mapping First, we need to map the given Process Tree onto the general DES model components: process (system), entities, simulation activities and events [13]. The Process Tree represents the process or system to be simulated. The entities are process instances, i.e. a run from start to end through the Process Tree. Simulation activities represent leaf and operator nodes. Each leaf node n has one corresponding simulation activity denoted as ϕ_n^{act}. Each operator node n, except type 'sequence', has two simulation activities: one split activity ϕ_n^{split} and one join activity ϕ_n^{join}. The start and end of a simulation activity constitutes one or more events. A simulation activity of a leaf node has exactly one start and one end event. A split activity has one start event and multiple end events, one end event for each outgoing branch of the corresponding operator node. A join activity has multiple start events, one start event for each outgoing branch of the corresponding operator node, and one end event.

On line 8 Algorithm 1 starts iterating over all nodes of the tree. For each leaf node it adds the simulation activity ϕ_n^{act} to the set of simulation activities. The simulation activity ϕ_n^{act} has a property *end* which points to the end event of that activity. For an operator node that is not of the sequence type, the algorithm adds a split activity ϕ_n^{split} and one join activity ϕ_n^{join} to the set of simulation activities. The split activity saves pointers to as much end events as its corresponding operator node has children. The join activity has a pointer to its single end event.

Linking nodes After the traversal of the tree, Algorithm 1 is going to link the events together to form a simulation series of events. The algorithm on line 23-25 sets the start event(s) of an activity $\phi_{n'}^m$ equal to the end event(s) of previous activities ϕ_n^m. In this way, if an end event of an activity ϕ_n^m happened, it will start another activity $\phi_{n'}^m$.

Algorithm 2 determines to which end event(s) the start event of activity ϕ_n^m will be linked. The start event of split and leaf node simulation activities depend on the operator semantics of the parent. If the parent is a 'sequence', the start event is the end of the previous child or the parent in case of the first child. If the parent is a 'choice', 'parallel' or 'or', the start event is one of the parent split end events. If the parent is a 'loop', the first child starts with the join of the 'loop', the second and third child start with one of the 'loop' split end events. Join activities start with the end events of all the incoming branches. The activities linked to the root node of the tree get the arrival as a (one of) the start event(s).

Execute simulation model Once all the simulation activities are linked to each other, Algorithm 1 simulates t entities on lines 26-28. For each entity, the algorithm first executes the simulation activity that starts right after the arrival of the entity, i.e. event e_0. To determine that simulation activity, it uses the *Match* function, that given an event, retrieves the simulation activity with that event as start event. The *Execute* function will process the simulation activity according to the semantics of the corresponding node type. A 'choice' split activity executes only one of its child branches. A 'parallel' split activity executes all of its child branches concurrently. An 'or' split activity executes one or more of its child branches concurrently. A 'loop' split activity executes either the second (redo) or third (exit) child branch. Because each split activity has a matching join of the same type, the join activity merges each of the executed child branches. The execution of activities linked to visible leaf nodes in the tree are logged, i.e. the label of the activity is added to the current trace.

Finally, Algorithm 1 will add noise to the traces if the noise probability Π^{Noise} is higher than zero. Only traces with more than one activity can be selected for noise insertion. The inserted noise is of a random type: remove head, remove body, remove tail, swap activities and remove activities (see Definition 3). The algorithm returns the final log with t traces.

Example To illustrate the simulation algorithm (see Algorithm 1) described above, consider the Process Tree PT_1 in Fig. 3. The first step maps all of the tree nodes to simulation activities and connects these activities to (an) end event(s). The output of the first step is shown in the first three columns of Table 1. The root node is a sequence node which has no corresponding simulation activity. For each of the leaf nodes $a, b, c, d, e, f, g, h, i, j, k$ a simulation activity with one end event is added. For example, leaf node 'a' leads to the simulation activity ϕ_a^{act} with end event e_a^{end}. Each of the remaining operator nodes $\wedge, \times, \circlearrowleft, \vee$ leads to two simulation activities, one for the split and one for the join. The split simulation activity of node $\wedge$, for example, has two end events $\{e_{\wedge_1}^{end}, e_{\wedge_2}^{end}\}$, one for each outgoing branch. The join simulation activity has a single end event $e_{\wedge_{join}}^{end}$.

The second step links the simulation activities to each other by determining the start events for each activity. The output of this step can be seen in column 4 of Table 1. The process starts with activity 'a', i.e. the start event of ϕ_a^{act} is the arrival of an entity e_0. The split simulation activity $\phi_\wedge^{split}$ directly follows activity 'a' and therefore has e_a^{end} as its start event. After the split, activity 'b' and 'c' are executed in parallel and synchronized by the join $e_{\wedge_{join}}^{end}$ that has the end of both activities as start events, i.e. $\{e_b^{end}, e_c^{end}\}$. Note that the loop differs from the other operator nodes. A loop starts with a join that merges the end of the choice join and the end of activity 'h' $\{e_{\times_{join}}^{end}, e_h^{end}\}$. Then activity 'g' is executed, followed by the loop split $\phi_\circlearrowleft^{split}$. After the loop split, either the second or third child can start.

In the third step, the simulation runs for t times, each time generating one trace. Algorithm 1 starts by executing the simulation activity ϕ_a^{act} that starts

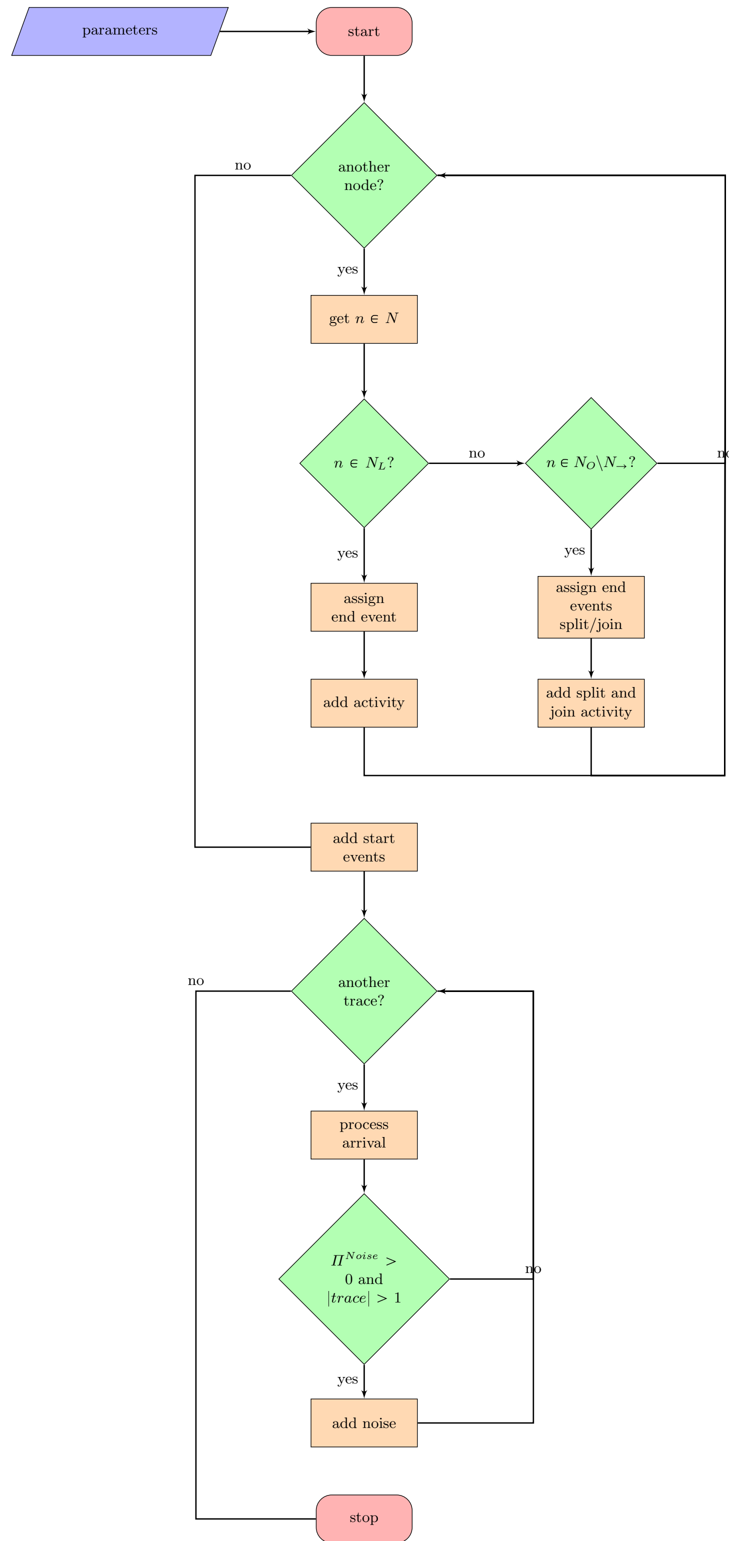


Fig. 2: Flowchart of Simulation Algorithm

Algorithm 1 : Simulate Process Tree into event log

1: **Input:**
2: PT: Process Tree
3: t: the number of traces
4: Π^{Noise}: the amount of noise
5: **Output:**
6: L: event log
7: $\Phi = \{\}$ ▷set of simulation activities
8: **for** $n \in PT$ **do** ▷create a simulation activity for each node
9: **if** $n \in N_L$ **then**
10: $\phi_n^{act}.end \leftarrow e_n^{end}$ ▷create simulation activity and assign end event
11: $\Phi \leftarrow \Phi \cup \phi_n^{act}$ ▷add activity to set of simulation activities
12: **else if** $n \in N_O \backslash \{N_\rightarrow\}$ **then**
13: **if** $n \in N_\circlearrowleft$ **then**
14: $\phi_n^{split}.end \leftarrow \{e_{n_{redo}}^{end}, e_{n_{exit}}^{end}\}$ ▷create simulation split activity and assign end events
15: **else**
16: $\phi_n^{split}.end \leftarrow \{e_{n_i}^{end} | i \in [1, |c(n)|]\}$ ▷create simulation split activity and assign end events
17: **end if**
18: $\Phi \leftarrow \Phi \cup \phi_n^{split}$ ▷add split to set of simulation activities
19: $\phi_n^{join}.end \leftarrow e_{n_{join}}^{end}$ ▷create simulation join activity and assign end event
20: $\Phi \leftarrow \Phi \cup \phi_n^{join}$ ▷add join to set of simulation activities
21: **end if**
22: **end for**
23: **for** $\phi_n^m \in \Phi$ **do**
24: $\phi_n^m.start \leftarrow GetStartEvent(n, m)$ ▷determine start event(s) of simulation activity
25: **end for**
26: **for** $i \in [1, t]$ **do** ▷create t entities
27: $\sigma_i \leftarrow \langle\rangle$ ▷start with an empty trace
28: $Execute(Match(e_0, \Phi), E, \sigma_i)$ ▷process arrival of entity i
29: **if** $\Pi^{Noise} > 0$ **then**
30: $x \leftarrow random(0, 1)$
31: **if** $|\sigma_i| > 1$ and $x < \Pi^{Noise}$ **then** ▷exclude traces of length one
32: type ← random(head,body,tail,swap,remove)
33: add noise type to σ_i ▷see Definition 3
34: **end if**
35: **end if**
36: $L \leftarrow L \cup \sigma_i$ ▷add trace of entity i to the log
37: **end for**
38: **return** L

Algorithm 2 : Get start event

1: **Input:**
2: n: node in Process Tree
3: m: type in act, split, join
4: **Start GetStartEvent(n,m)**
5: start ← {}
6: **if** $n = r$ **then** ▷if the node is the root
7: **if** $m = join\ and\ n \notin N_{\circlearrowleft}$ **then**
8: $start \leftarrow \{GetEndEvent(c(n)_1, \ldots, GetEndEvent(c(n)_{|c(n)|})\}$
9: **else if** $m = join\ and\ n \in N_{\circlearrowleft}$ **then**
10: $start \leftarrow \{e_0, GetEndEvent(c(n)_2)\}$
11: **else if** $m = split\ and\ n \in N_{\circlearrowleft}$ **then**
12: $start \leftarrow GetEndEvent(c(n)_1)$
13: **else**
14: $start \leftarrow e_0$ ▷e_0: arrival of an entity
15: **end if**
16: **else if** $m = join\ and\ n \notin N_{\circlearrowleft}$ **then** ▷for all joins except joins in loops
17: $start \leftarrow \{GetEndEvent(c(n)_1, \ldots, GetEndEvent(c(n)_{|c(n)|})\}$
18: **else if** $m = split\ and\ n \in N_{\circlearrowleft}$ **then**
19: $start \leftarrow GetEndEvent(c(n)_1)$
20: **else if** $p(n) \in N_{\rightarrow}$ **then** ▷parent is a sequence
21: **if** $n = c(p(n))_1$ **then** ▷first child in a sequence
22: **if** $m = join\ and\ n \in N_{\circlearrowleft}$ **then**
23: $start \leftarrow \{GetStartEvent(p(n)), GetEndEvent(c(n)_2)\}$
24: **else**
25: $start \leftarrow GetStartEvent(p(n))$
26: **end if**
27: **else** ▷second or further child in a sequence
28: **if** $m = join\ and\ n \in N_{\circlearrowleft}$ **then**
29: $start \leftarrow \{GetEndEvent(c(p(n))_{i-1}), GetEndEvent(c(n)_2)\}$
30: **else**
31: $start \leftarrow GetEndEvent(c(p(n))_{i-1})$ ▷start event equals the end event of previous child
32: **end if**
33: **end if**
34: **else if** $p(n) \in N_O$ where $o \in \{\times, \wedge, \vee\}$ **then**
35: **if** $m = join\ and\ n \in N_{\circlearrowleft}$ **then**
36: $start \leftarrow \{e^{end}_{p(n)_{split} \rightarrow n_{join}}, GetEndEvent(c(n)_2)\}$
37: **else**
38: $start \leftarrow e^{end}_{p(n)_{split} \rightarrow n}$ ▷specific end event of split parent operator
39: **end if**
40: **else** ▷parent is a loop
41: **if** $n = c(p(n))_1$ **then** ▷first child in a loop
42: **if** $m = join\ and\ n \in N_{\circlearrowleft}$ **then**
43: $start \leftarrow \{e^{end}_{p(n)_{join} \rightarrow n_{join}}, GetEndEvent(c(n)_2)\}$
44: **else**
45: $start \leftarrow e^{end}_{p(n)_{join}}$
46: **end if**
47: **else** ▷second or third child in a loop
48: **if** $m = join\ and\ n \in N_{\circlearrowleft}$ **then**
49: $start \leftarrow \{e^{end}_{p(n)_{split} \rightarrow n_{join}}, GetEndEvent(c(n)_2)\}$
50: **else**
51: $start \leftarrow e^{end}_{p(n)_{split} \rightarrow n}$ ▷specific end event of split parent loop
52: **end if**
53: **end if**
54: **end if**
55: **return** start

Algorithm 3 : Get end event

```
1: Input:
2:     n: node in Process Tree
3: Output:
4:     end: end event of node n
5: Start GetEndEvent(n)
6: if n ∈ N_L then
7:     end ← e_n^end
8: else
9:     if n ∈ N_→ then
10:        end ← GetEndEvent(c(n)_|c(n)|)          ▻end event of last child
11:    else if n ∈ N_O where o ∈ {×, ∧, ∨} then
12:        end ← e^end_{n_join→p(n)}               ▻join of the parent operator
13:    else                                        ▻node has type loop
14:        end ← GetEndEvent(c(n)_3)               ▻end event of last child loop
15:    end if
16: end if
17: return end
```

Algorithm 4 : Match simulation activity

```
1: Input:
2:     e: event
3:     Φ: list of simulation activities
4: Output:
5:     φ_n^m: simulation activity
6: Start Match(e,Φ)
7: for φ_n^m ∈ Φ do
8:     if e ∈ φ_n^m.start then
9:         return φ_n^m
10:    end if
11: end for
```

Algorithm 5 : Execute simulation activity

```
1: Input:
2:     φ_n^m: simulation activity
3:     E: events that happened
4:     trace: trace to log
5: Output:
6:     trace: trace to log
7: Start Execute(φ_n^m,E,trace)
8: if m = split then
9:     if n ∈ N_× then
10:        i ← random(1, |c(n)|)
11:        Execute(Match(e_{n_i}^{end}, Φ), E, trace)
12:        E ← E ∪ e_{n_i}^{end}
13:    else if n ∈ N_∧ then
14:        for i ∈ [1, |c(n)|] do
15:            Execute(Match(e_{n_i}^{end}, Φ), E, trace)
16:            E ← E ∪ e_{n_i}^{end}
17:        end for
18:    else if n ∈ N_∨ then
19:        j ← random(1, |c(n)|)
20:        for i ∈ [1, j] do
21:            Execute(Match(e_{n_i}^{end}, Φ), E, trace)
22:            E ← E ∪ e_{n_i}^{end}
23:        end for
24:    else if n ∈ N_↺ then
25:        i ← random(2, 3)
26:        Execute(Match(e_{n_i}^{end}, Φ), E, trace)
27:        E ← E ∪ e_{n_i}^{end}
28:    end if
29: else if m = join then
30:    if n ∈ {N_× ∪ N_↺} then
31:        Execute(Match(e_{n_join}^{end}, Φ), E, trace)
32:    else if n ∈ N_∧ then
33:        if φ_n^m.start ⊆ E then
34:            Execute(Match(e_{n_join}^{end}, Φ), E, trace)
35:        end if
36:    else if n ∈ N_∨ then
37:        if ∀i ∈ [1, j] : (e_{n_i}^{start}) ∈ E then
38:            Execute(Match(e_{n_join}^{end}, Φ), E, trace)
39:        end if
40:    end if
41:    E ← E ∪ e_{n_join}^{end}
42: else if m =act then
43:    if n ≠ τ then
44:        trace ← trace · m(n)                ▷add label of node n to trace
45:        Execute(Match(e_n^{end}, Φ), E, trace)
46:    end if
47:    E ← E ∪ e_n^{end}
48: end if
49: return trace
```

at the arrival of an entity. Activity ‘a’ is added to the current trace. Next, the *Match* function will retrieve the simulation activity $\phi_{\wedge}^{split}$ that starts when a has ended. The execution of $\phi_{\wedge}^{split}$ activates both ϕ_{b}^{act} and ϕ_{c}^{act} in random order. After both, activities ‘b’ and ‘c’ have ended, the join $\phi_{\wedge}^{join}$ synchronizes the parallel flows and triggers the choice split $\phi_{\times}^{split}$. An example execution is shown in Table 2.

In the final step Algorithm 1 adds noise to the generated trace. Consider the trace $\langle a, c, b, f, g, h, g, i, j \rangle$ as shown in the last row of Table 2. The swap task noise type would swap two random activities, e.g. ‘b’ and ‘f’ resulting in the noisy trace $\langle a, c, f, b, g, h, g, i, j \rangle$.

Tree node	**Simulation activity**	**End event(s)**	**Start event(s)**
$\rightarrow$	/	/	/
a	ϕ_{a}^{act}	e_{a}^{end}	e_0 (arrival)
$\wedge$	$\phi_{\wedge}^{split}$	$\{e_{\wedge 1}^{end}, e_{\wedge 2}^{end}\}$	e_{a}^{end}
	$\phi_{\wedge}^{join}$	$e_{\wedge join}^{end}$	$\{e_{b}^{end}, e_{c}^{end}\}$
b	ϕ_{b}^{act}	e_{b}^{end}	$e_{\wedge 1}^{end}$
c	ϕ_{c}^{act}	e_{c}^{end}	$e_{\wedge 2}^{end}$
$\times$	$\phi_{\times}^{split}$	$\{e_{\times 1}^{end}, e_{\times 2}^{end}, e_{\times 3}^{end}\}$	$e_{\wedge join}^{end}$
	$\phi_{\times}^{join}$	$e_{\times join}^{end}$	$\{e_{d}^{end}, e_{e}^{end}, e_{f}^{end}\}$
d	ϕ_{d}^{act}	e_{d}^{end}	$e_{\times 1}^{end}$
e	ϕ_{e}^{act}	e_{e}^{end}	$e_{\times 2}^{end}$
f	ϕ_{f}^{act}	e_{f}^{end}	$e_{\times 3}^{end}$
$\circlearrowleft$	$\phi_{\circlearrowleft}^{split}$	$\{e_{\circlearrowleft redo}^{end}, e_{\circlearrowleft exit}^{end}\}$	e_{g}^{end}
	$\phi_{\circlearrowleft}^{join}$	$e_{\circlearrowleft join}^{end}$	$\{e_{\times join}^{end}, e_{h}^{end}\}$
g	ϕ_{g}^{act}	e_{g}^{end}	$e_{\circlearrowleft join}^{end}$
h	ϕ_{h}^{act}	e_{h}^{end}	$e_{\circlearrowleft redo}^{end}$
i	ϕ_{i}^{act}	e_{i}^{end}	$e_{\circlearrowleft exit}^{end}$
$\vee$	$\phi_{\vee}^{split}$	$\{e_{\vee 1}^{end}, e_{\vee 2}^{end}\}$	e_{i}^{end}
	$\phi_{\vee}^{join}$	$e_{\vee join}^{end}$	$\{e_{j}^{end}, e_{k}^{end}\}$
j	ϕ_{j}^{act}	e_{j}^{end}	$e_{\vee 1}^{end}$
k	ϕ_{k}^{act}	e_{k}^{end}	$e_{\vee 2}^{end}$

Table 1: Output of mapping and linking steps

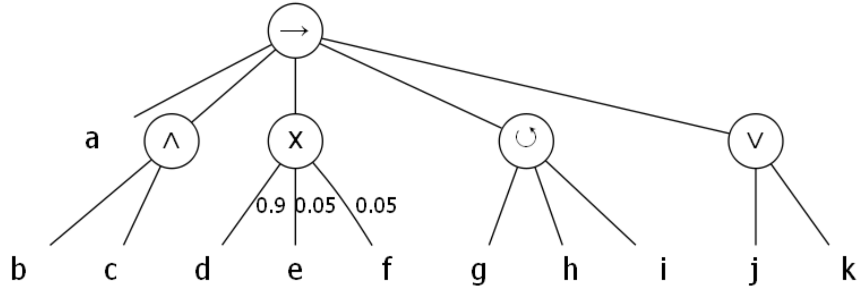


Fig. 3: Process Tree PT_1

Step	Simulation activity	Trace
1	ϕ_a^{act}	$\langle a \rangle$
2	$\phi_\wedge^{split}$	$\langle a \rangle$
3	ϕ_c^{act}	$\langle a, c \rangle$
4	ϕ_b^{act}	$\langle a, c, b \rangle$
5	$\phi_\wedge^{join}$	$\langle a, c, b \rangle$
6	$\phi_\times^{split}$	$\langle a, c, b \rangle$
7	ϕ_f^{act}	$\langle a, c, b, f \rangle$
8	$\phi_\times^{join}$	$\langle a, c, b, f \rangle$
9	$\phi_\circlearrowleft^{join}$	$\langle a, c, b, f \rangle$
10	ϕ_g^{act}	$\langle a, c, b, f, g \rangle$
11	$\phi_\circlearrowleft^{split}$	$\langle a, c, b, f, g \rangle$
12	ϕ_h^{act}	$\langle a, c, b, f, g, h \rangle$
13	$\phi_\circlearrowleft^{join}$	$\langle a, c, b, f, g, h \rangle$
14	ϕ_g^{act}	$\langle a, c, b, f, g, h, g \rangle$
15	$\phi_\circlearrowleft^{split}$	$\langle a, c, b, f, g, h, g \rangle$
16	ϕ_i^{act}	$\langle a, c, b, f, g, h, g, i \rangle$
17	$\phi_\vee^{split}$	$\langle a, c, b, f, g, h, g, i \rangle$
18	ϕ_j^{act}	$\langle a, c, b, f, g, h, g, i, j \rangle$
19	$\phi_\vee^{join}$	$\langle a, c, b, f, g, h, g, i, j \rangle$

Table 2: Example execution of PT_1

4 Implementation

The simulation of the DES model built in Section 3 is implemented using the SimPy simulation library [14]. The implementation is available as a Python application (see `https://github.com/tjouck/PTandLogGenerator`) and a ProM plugin [15].

4.1 Python Application

The Python application `generate_logs.py` is callable from commandline. It generates logs from given Process Trees in batches. The user can specify 5 arguments:

- size: the size of the event log in terms of number of traces t
- noise: the probability to add noise to a trace $\Pi^{Noise} \in [0, 1]$
- input folder: the location where the Process Trees are stored
- timestamps: whether to include timestamps for each activity
- format: generate a log in XES or CSV format

The first two arguments are mandatory, the last three are optional. The input folder is by default the '../data/trees/' folder in the project. The trees in this folder should be in the newick tree format.[1] The timestamps option indicates whether timestamps are logged for each activity. By default each activity has a duration from a random uniform distribution. As a result the event will contain two events for each executed activity: one start event and one end event, each with a timestamp. Finally, with the format option the user can opt to generate events logs in CSV format rather than the standard XES format [9] (default setting).

Fig. 4 shows the help option of the `generate_logs.py` application that shows how to use it and lists all the arguments with help information.

4.2 ProM Plugin

The ProM 'PTandLogGenerator' package contains the plugin `Generate Log Collection (with noise) from Process Trees`. This plugin takes as input a Newick Tree Collection, which is a list of Process Trees.[2] Before the actual simulation the plugin asks the user to specify the input parameters: the size of the generated logs t and the percentage of noise $\Pi^{Noise} \in [0, 1]$ (see Fig. 5). The plugin returns a set of event logs in XES format in the workspace (see Figure 6).

[1] All nodes have names and distances.

[2] The 'PTandLogGenerator' package also contains a plugin to convert a regular Process Tree object into a Newick Tree Collection.

```
$ python generate_logs.py --h
=======================================
0:00:00.000 - Start Program
=======================================

usage: generate_logs.py [-h] [--i [input_folder]] [--t [timestamps]]
                        [--f [format]]
                        size noise

Simulate event logs from process trees.

positional arguments:
  size                number of traces to simulate
  noise               probability to insert noise into trace

optional arguments:
  -h, --help          show this help message and exit
  --i [input_folder]  specify the relative address to the trees folder,
                      default=../data/trees/
  --t [timestamps]    indicate whether to include timestamps or not,
                      default=False
  --f [format]        indicate which format to use for the log: xes or csv,
                      default=xes
=======================================
0:00:00.006 - End Program
Elapsed time: 0:00:00.006
=======================================
```

Fig. 4: Help information of the `generate_logs.py` application

5 Conclusions and Future Work

Synthetic event logs are needed to test and fine-tune process mining techniques. Many simulation tools exist, but none of the current tools supports the input of Process Trees. Yet, Process Trees are an ideal input of a simulator as they are inherently sound and, as a result, never deadlock.

This paper introduced a simulation tool for Process Trees. It allows users to influence the resulting log characteristics by setting the number of process instances and the amount of noise in the final event log. The generated event logs are in the XES standard format. Moreover, the tool has been integrated within the ProM framework and therefore makes it suitable for conducting automated experiments on control-flow discovery techniques.

The current version of the tool is limited to the control-flow perspective of processes. Possible extensions of the simulation tool regard the inclusion of other process perspectives such as time and resources.

References

1. van der Aalst, W.: Process Mining: Data Science in Action. Springer (2016)
2. Di Ciccio, C., Bernardi, M.L., Cimitile, M., Maggi, F.M.: Generating event logs through the simulation of Declare models. In: Workshop on Enterprise and Organizational Modeling and Simulation, Springer (2015) 20–36
3. De Medeiros, A.A., Gnther, C.W.: Process mining: Using CPN tools to create test logs for mining algorithms. In: Proceedings of the sixth workshop on the practical use of coloured Petri nets and CPN tools (CPN 2005). Volume 576. (2005)

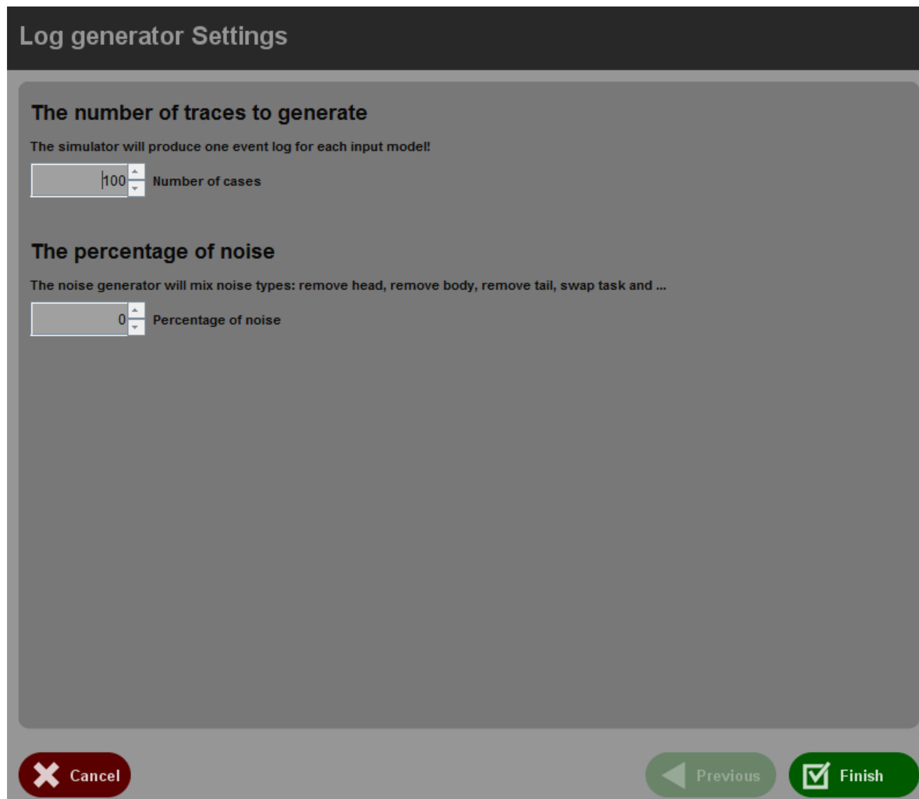


Fig. 5: The input dialog of the `Generate Log Collection (with noise) from Process Trees` plugin

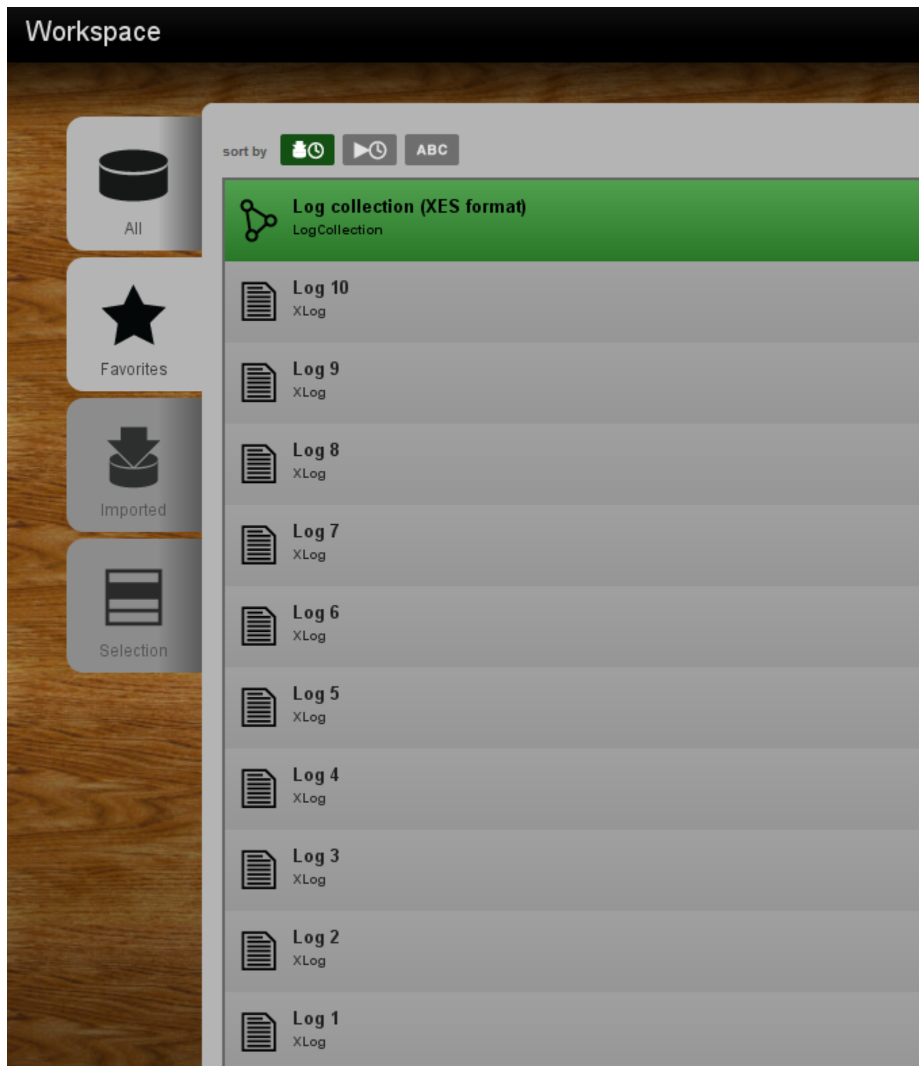


Fig. 6: The generated event logs in the workspace

4. Burattin, A.: PLG2: Multiperspective Processes Randomization and Simulation for Online and Offline Settings. Technical Report 1506.08415 (June 2015)
5. Vanden Broucke, S.K., Vanthienen, J., Baesens, B.: Straightforward Petri Net-based Event Log Generation in ProM. Available at SSRN 2489051 (2014)
6. van der Aalst, W., Buijs, J., Van Dongen, B.: Towards improving the representational bias of process mining. In: Data-Driven Process Discovery and Analysis. Springer (2012) 39–54 PM101 - 1.
7. Buijs, J.C.A.M.: Flexible Evolutionary Algorithms for Mining Structured Process Models. PhD thesis, Technische Universiteit Eindhoven, Eindhoven (2014)
8. Leemans, S.J., Fahland, D., van der Aalst, W.M.: Discovering block-structured process models from event logs-a constructive approach. In: Application and Theory of Petri Nets and Concurrency. Springer (2013) 311–329
9. Verbeek, H.M.W., Buijs, J.C.A.M., Van Dongen, B.F., Van Der Aalst, W.M.P.: Xes, xesame, and prom 6. In Soffer, P., Proper, E., eds.: Information Systems Evolution, Springer (2011) 60–75
10. de Medeiros, A.K.A., Weijters, A.J., van der Aalst, W.M.: Genetic process mining: an experimental evaluation. Data Mining and Knowledge Discovery **14**(2) (2007) 245–304
11. Maruster, L.: A machine learning approach to understand business processes. PhD thesis, Technische Universiteit Eindhoven, Eindhoven (2003)
12. Robinson, S.: Simulation: the practice of model development and use. Palgrave Macmillan (2014)
13. Shannon, R.E.: Introduction to simulation languages. In: Proceedings of the 9th conference on Winter simulation-Volume 1, Winter Simulation Conference (1977) 14–20
14. Matloff, N.: Introduction to discrete-event simulation and the simpy language. Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August **2** (2008) 2009
15. Jouck, T., Depaire, B.: PTandLogGenerator: a Generator for Artificial Event Data. In: Proceedings of the Demo Session of the 14th International Conference on Business Process Management (BPM 2016), Rio de Janeiro, Springer (2016)