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 on Process Trees, event logs and noise. In the next section, 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 (b) functions.

Definition 1 (Process Tree). Let $A \subseteq A$ be a finite set of activities and $PT$ be a tree: $PT = (N, r, c, p, s, b)$, 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
- $O = \{\to, \times, \land, \bigcup, \lor\}$ 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 $\tau$ 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:
  
  $$c(n) = \emptyset \text{ if } n \in N_L$$
  
  $$c(n) \in N^* \text{ if } n \in N_O$$
  
  such that:
  
  - each node except the root node has exactly one parent:
    
    $$\forall n \in N \setminus \{r\} : \exists p \in N_O : n \in c(p) \land \exists q \in N_O : p \neq q \land n \in c(q);$$
  
  - the root node has no parent:
    
    $$\exists n \in N : r \in c(n);$$
  
  - each node appears only once in the list of children of its parent:
    
    $$\forall n \in N : \forall i < j < c(n) : c(n)_i \neq c(n)_j;$$
  
  - a node with a loop operator type has exactly three children:
    
    $$\forall n \in N : (m(n) = \bigcup) \Rightarrow |c(n)| = 3.$$

- $p : N \rightarrow N$ is the parent relation function:
  
  $$p(n) = k \iff n \in c(k)$$

- each node has a probability of being chosen:
  
  $b : N \rightarrow [0, 1]$ is a mapping function mapping a probability to each node $n$:
  
  $$b(n) = \begin{cases} 1, & \text{if } p(n) \notin N_x \\
\in [0, 1], & \text{such that } \sum_{k \in c(p(n))} b(k) = 1 \text{ if } p(n) \in N_x. \end{cases}$$
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: $n = t(n_1, \ldots, n_k)$ 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 $\tau$ 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 $A \subseteq A$ be a finite set of activities. A trace $\sigma_j \in A^*$ is a sequence of activities. A log $L \in B(A^*)$ 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 $\sigma_j$ with $i \in [1, \frac{n}{2}]$
- Missing body: removes all activities $a_i$ in $\sigma_j$ with $i \in [\left(\frac{n}{2}\right) + 1, \frac{3n}{2}]$
- Missing tail: removes all activities $a_i$ in $\sigma_j$ with $i \in [\left(\frac{3n}{2}\right) + 1, n]$
- Swap tasks: interchanges two random activities $a_i$ and $a_j$ in $\sigma_j$ with $i \neq j$
- Remove task: remove random activity $a_i$ in $\sigma_j$

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 $\sigma_j$. The amount of noise in a log is another simulation parameter. The number of noisy traces follows a binomial distribution, i.e. $t' \sim \text{Binomial}(|L|, P^{\text{Noise}})$ with $t' \leq t$. $P^{\text{Noise}}$ 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 a 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 \( \phi^\text{act}_n \). Each operator node \( n \), except type ‘sequence’, has two simulation activities: one split activity \( \phi^\text{split}_n \) and one join activity \( \phi^\text{join}_n \). 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 \( \phi^\text{act}_n \) to the set of simulation activities. The simulation activity \( \phi^\text{act}_n \) has a property \( \text{end} \) 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 \( \phi^\text{split}_n \) and one join activity \( \phi^\text{join}_n \) 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 \( \phi^m_n \) equal to the end event(s) of previous activities \( \phi^m_m \). In this way, if an end event of an activity \( \phi^m_n \) happened, it will start another activity \( \phi^m_n \).

Algorithm 2 determines to which end event(s) the start event of activity \( \phi^m_n \) 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 \textit{Match} function, that given an event, retrieves the simulation activity with that event as start event. The \textit{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 \( \Pi_{\text{Noise}} \) 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 \( \phi_{\text{act}}^a \) with end event \( e_{\text{end}}^a \). Each of the remaining operator nodes \( \land, \times, \lor, \lor \) leads to two simulation activities, one for the split and one for the join. The split simulation activity of node \( \land \), for example, has two end events \( e_{\text{end}}^1 \) and \( e_{\text{end}}^2 \), one for each outgoing branch. The join simulation activity has a single end event \( e_{\text{end}}^{\land, \land} \).

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 \( \phi_{\text{act}}^a \) is the arrival of an entity \( e_0 \). The split simulation activity \( \phi_{\text{split}}^a \) directly follows activity ‘a’ and therefore has \( e_{\text{end}}^a \) as its start event. After the split, activity ‘b’ and ‘c’ are executed in parallel and synchronized by the join \( e_{\text{end}}^{\land, \land} \) that has the end of both activities as start events, i.e. \( \{ e_{\text{end}}^b, e_{\text{end}}^c \} \). 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’ \( \{ e_{\text{end}}^{\land, \land}, e_{\text{end}}^h \} \). Then activity ‘g’ is executed, followed by the loop split \( \phi_{\text{split}}^g \). 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 \( \phi_{\text{act}}^a \) that starts
Simulating PT with DES

parameters

start

another node?

yes

get $n \in N$

$n \in N_{\ell}$?

no

$n \notin N_{\ell}\setminus N_{\tau}$?

no

assign end event

add activity

add start events

add split and join activity

another trace?

yes

process arrival

$\Pi^{\text{Noise}} \geq 0$ and $\Pi^{\text{Noise}} > 1$

no

no

yes

add noise

stop

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: \(\Pi^{Noise}\): the amount of noise
5: **Output:**
6: \(L\): event log
7: \(\Phi\): set of simulation activities
8: for \(n \in PT\) do
9: \(\text{if } n \in N_L \text{ then}
10: \(\phi_{n\text{.end}} \leftarrow e_n^{end}\) \(\triangleright \text{create simulation activity and assign end event}
11: \(\Phi \leftarrow \Phi \cup \phi_{n\text{.end}}\) \(\triangleright \text{add activity to set of simulation activities}
12: \text{else if } n \in N_O \text{ then}
13: \(\text{if } n \in N_O \text{ then}
14: \(\phi_{n\text{split}.end} \leftarrow e_n^{end}\) \(\triangleright \text{create simulation split activity and assign end events}
15: \text{else}
16: \(\phi_{n\text{join}.end} \leftarrow e_n^{end}\) \(\triangleright \text{create simulation join activity and assign end event}
17: \text{end if}
18: \text{end if}
19: \text{end for}
20: for \(e_n^{end} \in \Phi\) do
21: \(\text{GetStartEvent}(n, m)\) \(\triangleright \text{determine start event(s) of simulation activity}
22: \text{end for}
23: for \(i \in [1, t]\) do
24: \(\text{Start with an empty trace}
25: \text{Execute}(\text{Match}(e_0, \Phi), E, \sigma_i)\) \(\triangleright \text{process arrival of entity i}
26: \text{if } \Pi^{Noise} > 0 \text{ then}
27: \(x \leftarrow \text{random}(0, 1)\)
28: \text{if } |\sigma_i| > 1 \text{ and } x < \Pi^{Noise} \text{ then}
29: \(\text{add noise type to } \sigma_i\) \(\triangleright \text{see Definition}\)
30: \text{end if}
31: \text{end if}
32: \(L \leftarrow L \cup \sigma_i\) \(\triangleright \text{add trace of entity i to the log}
33: \text{end for}
34: \text{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 \(\leftarrow\) \{
6: if \( n = r \) then \(\triangleright\) if the node is the root
7: if \( m = \text{join} \) and \( n \notin N_{3} \) then
8: start \(\leftarrow\) \{GetEndEvent\((c(n)_1,\ldots,\text{GetEndEvent}(c(n)_{|p(n)|})\)\}
9: else if \( m = \text{join} \) and \( n \in N_{3} \) then
10: start \(\leftarrow\) \{c_0, \text{GetEndEvent}(c(n)_2)\}
11: else if \( m = \text{split} \) and \( n \in N_{3} \) then
12: start \(\leftarrow\) GetEndEvent\((c(n)_1)\)
13: else
14: start \(\leftarrow c_0 \) \(\triangleright\) \(c_0\): arrival of an entity
15: end if
16: else if \( m = \text{join} \) and \( n \notin N_{3} \) then \(\triangleright\) for all joins except joins in loops
17: start \(\leftarrow\) \{GetEndEvent\((c(n)_1,\ldots,\text{GetEndEvent}(c(n)_{|p(n)|})\)\}
18: else if \( m = \text{split} \) and \( n \in N_{3} \) then
19: start \(\leftarrow\) GetEndEvent\((c(n)_1)\)
20: else if \( p(n) \in N_{\ldots} \) then \(\triangleright\) parent is a sequence
21: if \( n = c(p(n))_1 \) then \(\triangleright\) first child in a sequence
22: if \( m = \text{join} \) and \( n \notin N_{3} \) then
23: start \(\leftarrow\) \{GetStartEvent\((p(n))\), GetEndEvent\((c(n)_2)\)\}
24: else
25: start \(\leftarrow\) GetStartEvent\((p(n))\)
26: end if
27: else \(\triangleright\) second or further child in a sequence
28: if \( m = \text{join} \) and \( n \notin N_{3} \) then
29: start \(\leftarrow\) \{GetEndEvent\((c(p(n))_{|p(n)|-1})\), GetEndEvent\((c(n)_2)\)\}
30: else
31: start \(\leftarrow\) GetEndEvent\((c(p(n))_{|p(n)|})\) \(\triangleright\) start event equals the end event of previous child
32: end if
33: end if
34: else if \( p(n) \in N_{0} \) where \( o \in \{x, \land, \lor\} \) then
35: if \( m = \text{join} \) and \( n \notin N_{3} \) then
36: start \(\leftarrow\) \{c_{p(n)}^{\text{end}}_\text{split} \rightarrow_{n_{\text{join}}} \text{GetEndEvent}(c(n)_2)\}
37: else
38: start \(\leftarrow\) c_{p(n)}^{\text{end}}_\text{split} \rightarrow_{n} \(\triangleright\) specific end event of split parent operator
39: end if
40: else \(\triangleright\) parent is a loop
41: if \( n = c(p(n))_1 \) then \(\triangleright\) first child in a loop
42: if \( m = \text{join} \) and \( n \notin N_{3} \) then
43: start \(\leftarrow\) \{c_{p(n)}^{\text{end}}_\text{join} \rightarrow_{n_{\text{join}}} \text{GetEndEvent}(c(n)_2)\}
44: else
45: start \(\leftarrow\) c_{p(n)}^{\text{end}}_\text{join} \(\triangleright\) specific end event of join operator
46: end if
47: else \(\triangleright\) second or third child in a loop
48: if \( m = \text{join} \) and \( n \notin N_{3} \) then
49: start \(\leftarrow\) \{c_{p(n)}^{\text{end}}_\text{split} \rightarrow_{n_{\text{join}}} \text{GetEndEvent}(c(n)_2)\}
50: else
51: start \(\leftarrow\) c_{p(n)}^{\text{end}}_\text{split} \rightarrow_{n} \(\triangleright\) 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: \( \text{end} \): end event of node \( n \)
5: Start GetEndEvent(\( n \))
6: if \( n \in N_L \) then
7: \( \text{end} \leftarrow e_{\text{end}}^n \)
8: else
9: if \( n \in N_\rightarrow \) then
10: \( \text{end} \leftarrow \text{GetEndEvent}(e(n)_{|p}|) \) \(\triangleright\) end event of last child
11: else if \( n \in N_O \) where \( o \in \{\&, \land, \lor\} \) then
12: \( \text{end} \leftarrow e_{\text{end}}^{n_{\rightarrow o \rightarrow p(n)}} \) \(\triangleright\) join of the parent operator
13: else
14: \( \text{end} \leftarrow \text{GetEndEvent}(e(n)_3) \) \(\triangleright\) end event of last child loop
15: end if
16: end if
17: return \( \text{end} \)

Algorithm 4: Match simulation activity
1: Input:
2: \( e \): event
3: \( \Phi \): list of simulation activities
4: Output:
5: \( \phi_m^n \): simulation activity
6: Start Match(\( e, \Phi \))
7: for \( \phi_m^n \in \Phi \) do
8: if \( e \in \phi_m^n \_\text{start} \) then
9: return \( \phi_m^n \)
10: end if
11: end for
Algorithm 5 : Execute simulation activity

1: Input:
2: $\phi^m_n$: simulation activity
3: $E$: events that happened
4: $trace$: trace to log
5: Output:
6: $trace$: trace to log
7: Start Execute($\phi^m_n$, $E$, $trace$)
8: if $m = \text{split}$ then
9: if $n \in N_s$ then
10: $i \leftarrow \text{random}(1, \lceil c(n) \rceil)$
11: Execute($\text{Match}(e^e_{n_i}, \Phi), E, trace$)
12: $E \leftarrow E \cup e^e_{n_i}$
13: else if $n \in N_s$ then
14: for $i \in [1, \lceil c(n) \rceil]$ do
15: Execute($\text{Match}(e^e_{n_i}, \Phi), E, trace$)
16: $E \leftarrow E \cup e^e_{n_i}$
17: end for
18: else if $n \in N_s$ then
19: $j \leftarrow \text{random}(1, \lceil c(n) \rceil)$
20: for $i \in [1, j]$ do
21: Execute($\text{Match}(e^e_{n_i}, \Phi), E, trace$)
22: $E \leftarrow E \cup e^e_{n_i}$
23: end for
24: else if $n \in N_\delta$ then
25: $i \leftarrow \text{random}(2, 3)$
26: Execute($\text{Match}(e^e_{n_i}, \Phi), E, trace$)
27: $E \leftarrow E \cup e^e_{n_i}$
28: end if
29: else if $m = \text{join}$ then
30: if $n \in \{ N_s \cup N_\delta \}$ then
31: Execute($\text{Match}(e^e_{n_{\text{join}}}, \Phi), E, trace$)
32: else if $n \in N_s$ then
33: if $\phi^m_n.\text{start} \in E$ then
34: Execute($\text{Match}(e^e_{n_{\text{join}}}, \Phi), E, trace$)
35: end if
36: else if $n \in N_s$ then
37: if $\forall i \in [1, j] : (e^e_{n_i}.\text{start}) \in E$ then
38: Execute($\text{Match}(e^e_{n_{\text{join}}}, \Phi), E, trace$)
39: end if
40: end if
41: $E \leftarrow E \cup e^e_{n_{\text{join}}}$
42: else if $m = \text{act}$ then
43: if $n \neq \tau$ then
44: $\text{trace} \leftarrow \text{trace} \cdot m(n)$ \Comment{add label of node $n$ to trace}$
45: Execute($\text{Match}(e^e_{n}, \Phi), E, trace$)
46: end if
47: $E \leftarrow E \cup e^e_n$
48: end if
49: return $\text{trace}$
at the arrival of an entity. Activity ‘a’ is added to the current trace. Next, the
Match function will retrieve the simulation activity $\phi^{\text{split}}_a$ that starts when $a$
has ended. The execution of $\phi^{\text{split}}_a$ activates both $\phi^{\text{act}}_b$ and $\phi^{\text{act}}_c$ in random order.
After both, activities ‘b’ and ‘c’ have ended, the join $\phi^{\text{join}}_a$ synchronizes the
parallel flows and triggers the choice split $\phi^{\text{split}}_a$. An example execution is shown
in Table 2.

In the final step Algorithm 1 adds noise to the generated trace. Consider the
trace $\langle a, c, b, f, g, h, i, j \rangle$ 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 $\langle a, c, f, b, g, h, g, i, j \rangle$.

<table>
<thead>
<tr>
<th>Tree node</th>
<th>Simulation activity</th>
<th>End event(s)</th>
<th>Start event(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>$a$</td>
<td>$\phi^a_{\text{act}}$</td>
<td>$e^a_\text{end}$</td>
<td>$e_0$ (arrival)</td>
</tr>
<tr>
<td>$\wedge$</td>
<td>$\phi^a_{\text{split}}$</td>
<td>${ e^a_{\text{end}}, e^a_{\text{end}} }$</td>
<td>$e^a_\text{end}$</td>
</tr>
<tr>
<td>$b$</td>
<td>$\phi^b_{\text{act}}$</td>
<td>$e^b_\text{end}$</td>
<td>$e^b_\text{end}$</td>
</tr>
<tr>
<td>$c$</td>
<td>$\phi^c_{\text{act}}$</td>
<td>$e^c_\text{end}$</td>
<td>$e^c_\text{end}$</td>
</tr>
<tr>
<td>$\times$</td>
<td>$\phi^x_{\text{split}}$</td>
<td>${ e^x_\text{end}, e^x_{\text{end}}, e^x_{\text{end}} }$</td>
<td>$e^x_\text{end}$</td>
</tr>
<tr>
<td>$d$</td>
<td>$\phi^d_{\text{act}}$</td>
<td>$e^d_\text{end}$</td>
<td>$e^d_\text{end}$</td>
</tr>
<tr>
<td>$e$</td>
<td>$\phi^e_{\text{act}}$</td>
<td>$e^e_\text{end}$</td>
<td>$e^e_\text{end}$</td>
</tr>
<tr>
<td>$f$</td>
<td>$\phi^f_{\text{act}}$</td>
<td>$e^f_\text{end}$</td>
<td>$e^f_\text{end}$</td>
</tr>
<tr>
<td>$\odot$</td>
<td>$\phi^\odot_{\text{split}}$</td>
<td>${ e^{\odot}<em>{\text{redo}}, e^{\odot}</em>{\text{exit}} }$</td>
<td>$e^{\odot}_{\text{end}}$</td>
</tr>
<tr>
<td>$g$</td>
<td>$\phi^g_{\text{act}}$</td>
<td>$e^g_{\text{end}}$</td>
<td>$e^g_{\text{end}}$</td>
</tr>
<tr>
<td>$h$</td>
<td>$\phi^h_{\text{act}}$</td>
<td>$e^h_{\text{end}}$</td>
<td>$e^h_{\text{end}}$</td>
</tr>
<tr>
<td>$i$</td>
<td>$\phi^i_{\text{act}}$</td>
<td>$e^i_{\text{end}}$</td>
<td>$e^i_{\text{end}}$</td>
</tr>
<tr>
<td>$\vee$</td>
<td>$\phi^\vee_{\text{split}}$</td>
<td>${ e^\vee_{\text{end}}, e^\vee_{\text{end}} }$</td>
<td>$e^\vee_{\text{end}}$</td>
</tr>
<tr>
<td>$j$</td>
<td>$\phi^j_{\text{act}}$</td>
<td>$e^j_{\text{end}}$</td>
<td>$e^j_{\text{end}}$</td>
</tr>
<tr>
<td>$k$</td>
<td>$\phi^k_{\text{act}}$</td>
<td>$e^k_{\text{end}}$</td>
<td>$e^k_{\text{end}}$</td>
</tr>
</tbody>
</table>

Table 1: Output of mapping and linking steps
Simulating PT with DES

**Fig. 3: Process Tree PT**

**Table 2: Example execution of PT**

<table>
<thead>
<tr>
<th>Step</th>
<th>Simulation activity</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\phi_{act}^a$</td>
<td>$\langle a \rangle$</td>
</tr>
<tr>
<td>2</td>
<td>$\phi_{split}^a$</td>
<td>$\langle a \rangle$</td>
</tr>
<tr>
<td>3</td>
<td>$\phi_{act}^c$</td>
<td>$\langle a, c \rangle$</td>
</tr>
<tr>
<td>4</td>
<td>$\phi_{act}^b$</td>
<td>$\langle a, c, b \rangle$</td>
</tr>
<tr>
<td>5</td>
<td>$\phi_{form}^b$</td>
<td>$\langle a, c, b \rangle$</td>
</tr>
<tr>
<td>6</td>
<td>$\phi_{split}^c$</td>
<td>$\langle a, c, b \rangle$</td>
</tr>
<tr>
<td>7</td>
<td>$\phi_{act}^x$</td>
<td>$\langle a, c, b, f \rangle$</td>
</tr>
<tr>
<td>8</td>
<td>$\phi_{form}^f$</td>
<td>$\langle a, c, b, f \rangle$</td>
</tr>
<tr>
<td>9</td>
<td>$\phi_{form}^c$</td>
<td>$\langle a, c, b, f \rangle$</td>
</tr>
<tr>
<td>10</td>
<td>$\phi_{act}^g$</td>
<td>$\langle a, c, b, f, g \rangle$</td>
</tr>
<tr>
<td>11</td>
<td>$\phi_{split}^g$</td>
<td>$\langle a, c, b, f, g \rangle$</td>
</tr>
<tr>
<td>12</td>
<td>$\phi_{act}^h$</td>
<td>$\langle a, c, b, f, g, h \rangle$</td>
</tr>
<tr>
<td>13</td>
<td>$\phi_{form}^h$</td>
<td>$\langle a, c, b, f, g, h \rangle$</td>
</tr>
<tr>
<td>14</td>
<td>$\phi_{act}^g$</td>
<td>$\langle a, c, b, f, g, h, g \rangle$</td>
</tr>
<tr>
<td>15</td>
<td>$\phi_{split}^g$</td>
<td>$\langle a, c, b, f, g, h, g \rangle$</td>
</tr>
<tr>
<td>16</td>
<td>$\phi_{act}^i$</td>
<td>$\langle a, c, b, f, g, h, g, i \rangle$</td>
</tr>
<tr>
<td>17</td>
<td>$\phi_{split}^i$</td>
<td>$\langle a, c, b, f, g, h, g, i \rangle$</td>
</tr>
<tr>
<td>18</td>
<td>$\phi_{act}^j$</td>
<td>$\langle a, c, b, f, g, h, g, i, j \rangle$</td>
</tr>
<tr>
<td>19</td>
<td>$\phi_{form}^j$</td>
<td>$\langle a, c, b, f, g, h, g, i, j \rangle$</td>
</tr>
</tbody>
</table>
4 Implementation

The simulation of the DES model built in Section 3 is implemented using the SimPy simulation library [4]. 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 $\Pi_{\text{Noise}} \in [0, 1]$
- 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 $\Pi_{\text{Noise}} \in [0, 1]$ (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.
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

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