Grammar-Based Process Model Representation for Probabilistic Conformance Checking

Akio Watanabe, Yousuke Takahashi, Hiroki Ikeuchi and Kotaro Matsuda*
NTT Network Service Systems Laboratories, NTT Corporation, Tokyo, Japan
Email: {akio.watanabe.ns, yousuke.takahashi.bs, hiroki.ikeuchi.re}@hco.ntt.co.jp, k_matsuda@fanfare-kk.com

***Abstract*—Probabilistic conformance checking methods, which use the probability of observed traces to evaluate the fitness between process models and event logs, have recently attracted much attention. In this paper, we propose a new process model representation, the Probabilistic Generative Process Model (PGPM), which can explicitly calculate the generation probabilities of traces in a process model. PGPM can explicitly and quickly compute the trace probabilities in a process model by expressing the generation procedure of traces on the basis of a probabilistic context-free grammar. PGPM enables us to apply probabilistic conformance checking to various process models. We also propose a probabilistic parameter estimation method based on the expectation-maximization (EM) algorithm to obtain a superior probabilistic process model that locally maximizes the likelihood for an event log.**

***Index Terms*—Probabilistic conformance checking, Probabilistic process model, Context-free grammar**

I. Introduction

Conformance checking is becoming increasingly important as opportunities for process mining practices increase. Conformance checking is the task of verifying that the traces in the observed event logs do not deviate from the business operations specified in the process model. A process model is a graph that defines the activities to be performed and the order of those activities in a business process. An event log is a set of traces, which are activity sequences executed by users. The recent advances in digital transformation have made it easier to record event logs. Therefore, when auditing whether a company operates by laws and regulations, many event logs are analyzed. As a result, auditors are increasingly using conformance checking.

Recently, probabilistic conformance checking (PCC) has attracted much attention. PCC is a method of comparing an event log with a probabilistic process model, which is a process model that can compute probabilities for arbitrary traces. The advantage of PCC is that it can treat the trace frequencies in the event log in model evaluation. For example, if there are two event logs $\{\langle a, b\rangle^{999}, \langle a, c\rangle^{1}\}$ and $\{\langle a, b\rangle^{1}, \langle a, c\rangle^{999}\}$, the appropriate model for each event log is different. Unlike usual conformance checking, where the evaluation results for these event logs are always the same, PCC can obtain different evaluation results depending on the trace frequencies.

However, several problems make PCC difficult to use. For PCC to be applicable, the probability of a trace appearing in the model and event log, i.e., the *trace probability*, must be obtainable. The trace probability is calculated as the sum of the probabilities of all transitions that can generate that trace in a model, i.e., the sum of the probabilities of all *alignments*. However, conventional probabilistic process models cannot explicitly obtain trace probabilities because the number of alignments is not finite in some models [1]. This problem makes PCC application difficult because the availability of PCC depends on the target model.

The other problem is that obtaining alignments is time-consuming. Existing methods either take long to search for all alignments or explore only some alignments at high speed.

In addition, existing probabilistic process discovery has the problem of estimation accuracy of trace probabilities. For PCC applications, a probabilistic process model must be obtained by estimating the probability parameters in a given process model from the event log. Although a method to obtain a probabilistic process model has been proposed, this existing method does not always maximize the likelihood of the model for the event logs.

This paper proposes a new probabilistic process model representation called Probabilistic Generative Process Model (PGPM). PGPM enables trace probabilities to always be calculable if the business process can be represented by a process tree. This trace probability is computed rapidly in accordance with dynamic programming. Furthermore, PGPM can use EM algorithm-based methods to estimate probability parameters. This estimation converges to the parameters that locally maximize the likelihood for the event log. Therefore, PGPM can obtain a more consistent model with the event log than models by conventional probabilistic process discovery. Thus, PGPM makes PCC methods widely applicable and improves their usefulness significantly.

We devised PGPM on the basis of the fact that a process tree, one of the traditional process model representations, can be transformed into a set of transformation procedures for a sequence of symbols called a *generation rule*. Since generation rules can formulate the procedures for generating traces, PGPM can obtain the generation probability of an arbitrary trace. This formulation is an analogy to the parsing approach in natural language processing. PGPM allows for the estimation of probability parameters and the inference of the generation procedure for traces. These estimation and inference algorithms, developed in natural language processing, are fast and can be applied to business processes that may have multiple instances of the same activity.

This paper is organized as follows. Section II describes examples of related research. Section III describes the mathematical problem set for this study. Section IV describes the proposed process model and the approach overview. Section

*He is now with Fanfare Inc., Tokyo 107-0052, Japan.

2022 4th International Conference on Process Mining (ICPM 2022), ISBN 979-8-3503-9714-7

V explains the conversion from an existing process model to PGPM. Section VI describes the methods to estimate probability parameters from event logs and inference of the trace generation procedure. Section VII presents the results of validating the effectiveness of PGPM through experiments. Section VIII presents conclusions and future work.

II. Related Work

Recently, PCC approaches have been proposed, which evaluate the conformance of models and logs on the basis of trace probabilities. Leemans et al. proposed a method to evaluate probabilistic process models by obtaining the trace probability distribution of an event log and a model and measuring the distance between the distributions on the basis of Earth Movers' Distance [1]. They also describe a similar model evaluation metric that measures the Entropy of event logs and models [2]. PCC provides a probabilistic process model and an event log as input and evaluates how close the trace probability distributions of the model and log are. Thus, PCC requires the trace probabilities in the probabilistic process model.

A probabilistic process model is obtained by extending a regular process model (without probabilities) to define the probability distribution of the following activity to be executed. Although Business Process Model and Notation (BPMN) or process trees [3] can be used to visualize process models, all existing PCCs use Stochastic Petri Nets (SPNs). SPNs are extensions of Petri Nets that can mathematically represent state transitions, with probabilities of transitions firing [5]. Various derivatives of SPNs with modified probability distributions have been proposed [6]. For example, there is the Generalized Stochastic Labeled Petri Net (GSLPN) [7], a model that includes activities in which events are not observed, called silent-activities. However, the method has been established to obtain trace probabilities for processes with repeated silent-activity loops (called silent-loops) by GSLPN [1]. This problem can occur with any SPN that handles a silent-activity. PGPM can be converted to an equivalent model that excludes the generation process corresponding to silent-activity through the normalization method so that this problem does not occur.

Rogge-Solti et al. defined a generically distributed transition SPN, an SPN plus a transition probability based on time intervals [8]. They also proposed a process discovery method that searches for its probabilistic parameters. Given a Petri Net and an event log, this method finds a plausible trace generation path in the model. It obtains probability parameters on the basis of the number of transitions in the path. However, this method uses only the most plausible path for estimating the probability parameters. If multiple paths correspond to a trace, this method can cause a suboptimal parameter estimation, which does not maximize the likelihood of the model for the event log. Since our method is based on the EM algorithm, the parameter estimation converges to values that locally maximize the likelihood of the model for the event log.

Conformance checking identifies whether a trace conforms to (can be generated from) a process model. Conformance checking can be classified into two approaches: alignment-based or token-based. In the alignment-based approach, the paths for traces in the model are estimated as alignments [9]–[12]. If there is an alignment, the model determines that the trace can be generated. In Adriansyah et al.' s approach [9], both the trace and the process model are represented by Petri Nets to obtain the alignment of two different datasets. In addition, obtained Petri Net can quantify the divergence between the model and the trace by defining a cost function. Boltenhagen et al. proposed a state-of-the-art method in alignment by adding a cost function with a discount parameter that can adjust the balance between calculation speed and accuracy [12]. Unlike the alignment-based approach, the token-based approach makes the token progress on the process model following the trace [13]. The model can generate a trace if the token can progress to finish states.

These methods sacrifice either speed or accuracy. The alignment-based approach accurately determines if the model and trace conform but takes much time. On the other hand, token-based approaches and state-of-the-art alignment methods [12] do not search all candidate paths to avoid a computational explosion. Therefore, they are fast but do not always find the correct path. PGPM uses an algorithm based on dynamic programming to efficiently calculate the probability of covering all alignments. This provides fast and accurate conformance checking.

III. Preliminaries

In process mining, event logs that record user behavior are used as observation data, and the event logs are analyzed using process models that represent potential regularities in user behavior. In this study, an event log is considered to be a multiset of traces. If the set of events representing the type of activity is $\mathcal{E}$, the trace can be represented as a series of events $\sigma \in \mathcal{E}^*$. We denote the trace by parentheses as $\sigma = \langle a, b, c \rangle (a, b, c \in \mathcal{E})$. $|\sigma|$ denotes the length of the trace (e.g., $|\langle a, b, c, b \rangle| = 4$). The inclusion of m traces σ in the event log L is denoted by $\{\sigma^m\}$ using superscripts. For example, it can be expressed as $L = \{\langle a, b, d \rangle^{100}, \langle a, c, d \rangle^1\}$.

In this study, we aim to express the occurrence probability of an event log L by a process model G. In other words, we aim to define a mathematical model G that represents $\Pr(L; G) = \prod_{\sigma^m \in L} \Pr(\sigma; G)^m$. This indicates *a likelihood* of G for L.

A process model is a graph showing user activities and their sequential relationships. It is generally represented by a BPMN, but here we formulate the process tree that will be our discussion point.

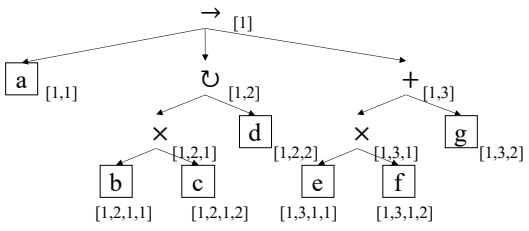


Fig. 1: Process tree

A process tree is an ordered tree that represents the dependencies of operators. Fig. 1 illustrates an example of a process

tree. A process tree has a set of leaf nodes N corresponding to activities that generate observation events and a set of internal nodes V representing connection relationships among child nodes by operators. Any node except the root node has one and only one node as its parent node. Because a process tree is an ordered tree, there is an order between nodes with the same parent node. In the illustration in this paper, we place the node with the smallest order on the left.

In this study, nodes in the process tree are identified by their addresses. The address indicates the order of each node from the root node to that node. Fig. 1 shows the address of each node. For example, the leaf node in event d has a root node ($\rightarrow$) and a second child node ($\circlearrowleft$) as ancestor nodes, and d itself is a second child node. Therefore, the address of the node at d is $[1, 2, 2]$.

Each node in the process tree has a corresponding operator. Operators indicate relationships between child nodes. This study uses four operators: SEQUENCE, SELECTION, PARALLEL, and LOOP. Each operator represents a symbol and relationship shown in Table I.

TABLE I: Operators in process tree

operator & symbol		description
SEQUENCE	$\rightarrow$	All child nodes are executed in order.
SELECTION	$\times$	Only one child node will be executed.
PARALLEL	$+$	All child nodes are executed in any order.
LOOP	$\circlearrowleft$	All child nodes are executed in order. repeatedly. After executing the first child node, the repeat stops probabilistically.
ACTIVITY	T	A convenience symbol indicating that a node is an activity.

A leaf node in the process tree has a correspondence $l(v)$ to one of the events, where $l(v) \in \mathcal{E} \cup \{\epsilon\}$. The ϵ is a special symbol indicating that the leaf node does not generate any events. Note that multiple nodes can have associations to the same event.

Here, we define some operations on nodes.

- $\mathrm{ch}(v, i)$: Obtain the i-th child node of the node v.
- $|v|$: Obtain the number of child nodes of the node v.
- $o(v)$: If the node v is an internal node, obtain the corresponding operator.
- $l(v)$: If the node v is a leaf node, obtain the corresponding event.
- $\oplus_{v,i}$: For a child node $\mathrm{ch}(v, i)$, obtain the symbol of the operator of the child node if it is an internal node, or obtain the symbol $T_{\mathrm{ch}(v,i)}$ if it is a leaf node. ($\oplus_{v,i} \in \{\rightarrow_{\mathrm{ch}(v,i)}, \times_{\mathrm{ch}(v,i)}, +_{\mathrm{ch}(v,i)}, \circlearrowleft_{\mathrm{ch}(v,i)}, T_{\mathrm{ch}(v,i)}\}$, where subscripts indicate addresses.)

IV. Model and Approach Overview

Our goal is to define a process model G for which the probability $\Pr(L; G)$ of occurrence of any trace σ is computable. To do so, we first formalize the procedure for generating traces. In the following, we use the process tree shown in Fig. 1 and the trace $\langle a, b, d, c, g, f \rangle$ generated from the tree as examples for the explanation.

If the events in a trace are classified hierarchically by their parent nodes, the inclusion relationships among events can be represented by a tree structure. First, for each event in the trace $\langle a, b, d, c, g, f \rangle$, the parent node in the process tree is examined. The parent nodes are $\rightarrow_{[1]}$, $\times_{[1,2,1]}$, $\circlearrowleft_{[1,2]}$, $\times_{[1,2,1]}$, $+_{[1,3]}$, and $\times_{[1,3,1]}$, respectively. If we search for the parent node for each parent node hierarchically, the parent-child relationship of the nodes can be represented by a tree structure as shown in Fig. 2. This tree showing the parent nodes of the trace is called a trace tree. In the trace tree, the symbol \$ is given as the parent node of the root node.

We defined the trace tree by exploring the parent node bottom-up in the explanation above. Conversely, looking at the trace tree top-down from the root node, the trace tree can be seen as the process by which the symbol \$ generates all the events in the trace through a series of transformations called *generation rules*. The transformation with generation rules equivalent to the trace tree in Fig. 2 is shown in Table II. In this example, one symbol A is repeatedly transformed into another symbol sequence α by the generation rule $A \Longrightarrow \alpha$.

The conversion process from a symbol to a symbolic sequence, such as in Table II, can be represented by using context-free grammar (CFG), which is used in natural language processing. CFG is a mathematical model for representing the grammatical structure of a sentence. In this study, traces are considered as sentences, and CFGs are used to represent the sub-process structure in traces. Formally, a CFG is represented by four pairs of elements.

- A set of non-terminal symbols M.
- A set of terminal symbols E.
- A set of generation rules R, where a generation rule is a transformation from one non-terminal symbol to another symbol sequence of the form $A \Longrightarrow \alpha (A \in M, \alpha \in \{M \cup E \cup \{\epsilon\}\}^*)$.
- The initial symbol is \$.

We found that process trees can be converted into equivalent CFG. The set of events $\mathcal{E}$ in a process tree is equivalent to the set of terminal symbols E. A node in the process tree corresponds to a non-terminal symbol represented by an operator. In addition, the generation rules correspond to the relationships between child nodes defined for each operator. For example, the generation rule $\rightarrow_{[1]} \Longrightarrow T_{[1,1]}, \circlearrowleft_{[1,2]}, +_{[1,3]}$ in Table II means that the SEQUENCE operator will execute three child nodes $T_{[1,1]}, \circlearrowleft_{[1,2]}, +_{[1,3]}$ in order. The correspondence between operators and generation rules will be described in Section V.

Furthermore, we propose an extension from a process tree to a probabilistic generative process model (PGPM) with trace generation probabilities. This extension is an application of the extension from CFG to Probabilistic Context-Free Grammar (PCFG). PCFG is the definition of CFG plus the set P of probabilities $P(A \Longrightarrow \alpha \in R)$ that any generative rule $A \Longrightarrow \alpha$ is applicable. Here, the total probability of the generating rules having the same non-terminal symbol A on the left side is 1. Similarly, PGPM is defined as $G = (M, E, R, \$, P)$ by adding the probability P of each generating rule to CFG.

Let $\mathbf{T}_\sigma$ be the set of possible trace trees for trace σ in model G. $\mathcal{T}_\sigma \in \mathbf{T}_\sigma$ is the sequence of generation rules up to generating σ. The probability of generating σ from G is $\Pr(\sigma; G) = \sum_{\mathcal{T}_\sigma \in \mathbf{T}_\sigma} \Pr(\mathcal{T}_\sigma) = \sum_{\mathcal{T}_\sigma \in \mathbf{T}_\sigma} \prod_{r \in \mathcal{T}_\sigma} P(r)$. In

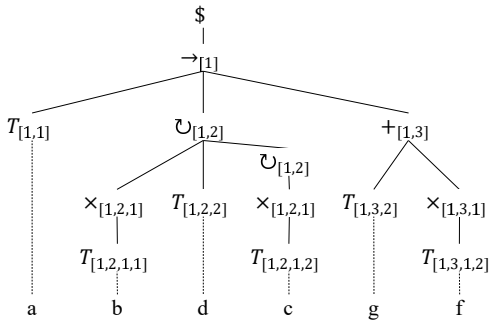


Fig. 2: Trace tree

TABLE II: Generation rules for trace tree in Fig. 2

executed generation rule r	generated symbol sequence	$P(r)$
$\$ \Longrightarrow \rightarrow_{[1]}$	$\langle \rightarrow_{[1]} \rangle$	1
$\rightarrow_{[1]} \Longrightarrow T_{[1,1]}, \circlearrowright_{[1,2]}, +_{[1,3]}$	$\langle T_{[1,1]}, \circlearrowright_{[1,2]}, +_{[1,3]} \rangle$	1
$T_{[1,1]} \Longrightarrow a$	$\langle a, \circlearrowright_{[1,2]}, +_{[1,3]} \rangle$	1
$\circlearrowright_{[1,2]} \Longrightarrow \times_{[1,2,1]}, T_{[1,2,2]}, \circlearrowright_{[1,2]}$	$\langle a, \times_{[1,2,1]}, T_{[1,2,2]}, \circlearrowright_{[1,2]}, +_{[1,3]} \rangle$	0.2
$\circlearrowright_{[1,2]} \Longrightarrow \times_{[1,2,1]}$	$\langle a, \times_{[1,2,1]}, T_{[1,2,2]}, \times_{[1,2,1]}, +_{[1,3]} \rangle$	0.8
$\times_{[1,2,1]} \Longrightarrow T_{[1,2,1,1]}$	$\langle a, T_{[1,2,1,1]}, T_{[1,2,2]}, \times_{[1,2,1]}, +_{[1,3]} \rangle$	0.6
$\times_{[1,2,1]} \Longrightarrow T_{[1,2,1,2]}$	$\langle a, T_{[1,2,1,1]}, T_{[1,2,2]}, T_{[1,2,1,2]}, +_{[1,3]} \rangle$	0.4
$T_{[1,2,1,1]} \Longrightarrow b$	$\langle a, b, T_{[1,2,2]}, T_{[1,2,1,2]}, T_{[1,2,1,2]}, +_{[1,3]} \rangle$	1
$T_{[1,2,2]} \Longrightarrow d$	$\langle a, b, d, T_{[1,2,1,2]}, +_{[1,3]} \rangle$	1
$T_{[1,2,1,2]} \Longrightarrow c$	$\langle a, b, d, c, +_{[1,3]} \rangle$	1
$+_{[1,3]} \Longrightarrow T_{[1,3,2]}, \times_{[1,3,1]}$	$\langle a, b, d, c, T_{[1,3,2]}, \times_{[1,3,1]} \rangle$	0.5
$T_{[1,3,2]} \Longrightarrow g$	$\langle a, b, d, c, g, \times_{[1,3,1]} \rangle$	1
$\times_{[1,3,1]} \Longrightarrow T_{[1,3,1,2]}$	$\langle a, b, d, c, g, T_{[1,3,1,2]} \rangle$	0.7
$T_{[1,3,1,2]} \Longrightarrow f$	$\langle a, b, d, c, g, f \rangle$	1

conclusion, any process tree can be converted to a PGPM $G = (M, E, R, \$, P)$ that can compute $\Pr(\sigma; G)$.

The procedure for analysis using PGPM is shown in Figure 3. The inputs are a process tree and an event log. The process tree is converted into a set of generation rules by the method described in Section V. At this time, ϵ-rules corresponding to silent-activities are removed by normalization. Next, the probability parameters of all rules are iteratively estimated to maximize the likelihood of the event log. The generated rules and probability parameters together are the PGPM. Using PGPM, we can infer a trace tree for a given arbitrary trace and its probabilities. Since parameter estimation and inference are very similar, we describe both in Section VI.

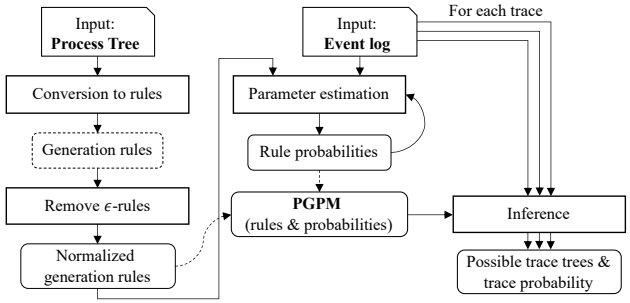


Fig. 3: Analysis procedure using PGPM

V. Conversion of Process Trees to Generation Rules

We describe a method for obtaining elements other than P in the PGPM from the process tree. As already mentioned, the set of terminal symbols E coincides with the set of events $\mathcal{E}$. Also, the initial symbol is always \$. Therefore, here we describe a method to obtain M and R from the process tree.

A. Conversion of SEQUENCE, LOOP, SELECTION, and ACTIVITY Operators

Non-terminal symbols correspond to each node in the process tree. Also, the generation rules depend only on the operator and child nodes of that node, except for the PARALLEL operator. For example, for a node with the SEQUENCE operator, the generation rule only needs to add a conversion from a non-terminal symbol indicating the operator to a non-terminal symbol indicating the child node ($\rightarrow_v \Longrightarrow \oplus_{v,1}, \ldots, \oplus_{v,|v|}$). Each of the different generation rules for each operator is shown in Table III. Note that in the generation rules obtained by the proposed method, there are no more than three symbols on the right side. This is a widely known method for applying the algorithm described below.

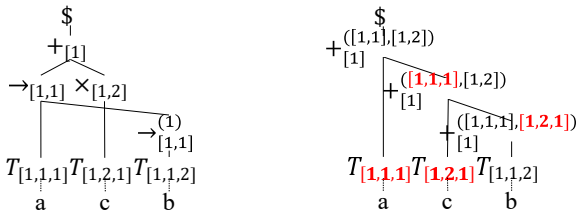
TABLE III: Generation rules for each operator (except PARALLEL)

$o(v)$	Added generation rules
$\rightarrow$	$\rightarrow_v \Longrightarrow \oplus_{v,1}, \rightarrow_v^{(1)}$, $\rightarrow_v^{(i-1)} \Longrightarrow \oplus_{v,i}, \rightarrow_v^{(i)}$ $(\forall i = 2 \ldots \lvert v \rvert)$, $\rightarrow_v^{(\lvert v \rvert - 1)} \Longrightarrow \oplus_{v,\lvert v \rvert}$.
$\circlearrowright$	$\circlearrowright_v \Longrightarrow \oplus_{v,1}, \circlearrowright_v^{(1)}$, $\circlearrowright_v \Longrightarrow \oplus_{v,1}$, $\circlearrowright_v^{(i-1)} \Longrightarrow \oplus_{v,i}, \circlearrowright_v^{(i)}$ $(\forall i = 2 \ldots \lvert v \rvert)$, $\circlearrowright_v^{(\lvert v \rvert - 1)} \Longrightarrow \oplus_{v,\lvert v \rvert}, \circlearrowright_v$.
$\times$	$\times_v \Longrightarrow \oplus_{v,i}$ $(\forall i = 1 \ldots \lvert v \rvert)$.
T	$T_v \Longrightarrow l(v)$.

B. Conversion of PARALLEL Operator

Unlike other operators, nodes with PARALLEL operators cannot be simply converted into generation rules due to *discontinuity*. We illustrate this problem using two trace trees in Fig. 4 for a trace $\langle a, c, b \rangle$ and a process tree in Fig. 5 that can generate the trace as an example. If we perform the conversion similar to the procedure in the previous section, the SEQUENCE operator for node $[1, 1]$ in this model is represented by two generation rules ($\rightarrow_{[1,1]} \Longrightarrow T_{[1,1,1]}, \rightarrow_{[1,1]}^{(1)}$) and ($\rightarrow_{[1,1]}^{(1)} \Longrightarrow T_{[1,1,2]}$). In this case, no matter what the order of transformation is, $T_{[1,1,1]}$ and $T_{[1,1,2]}$ are always adjacent as $\langle \ldots, T_{[1,1,1]}, T_{[1,1,2]}, \ldots \rangle$. However, to generate $\langle a, c, b \rangle$, $\times_{[1,2]}$ must be inserted between $T_{[1,1,1]}$ and $T_{[1,1,2]}$. If this is represented graphically, the edges intersect, as shown in Fig. 4a. This grammatical structure with intersecting edges is called *discontinuity* in linguistics, and the transformation cannot be expressed in a normal CFG.

Generation rules can represent the PARALLEL operator without discontinuity if the state of the sub-process for each subtree is preserved. Fig. 4b shows the generation process of $\langle a, c, b \rangle$ without discontinuity under the PARALLEL operator. In this generation rule, the "activity generated at each child node" is held by the superscript of the non-terminal symbol indicating the PARALLEL operator. For example, when the



(a) Discontinuity **(b)** Continuity

Fig. 4: Trace trees under PARALLEL operator

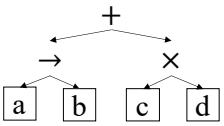


Fig. 5: Process tree with PARALLEL

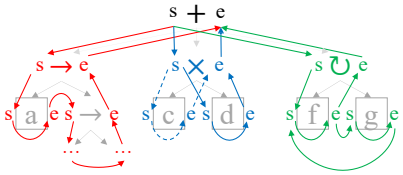


Fig. 6: States and transitions under PARALLEL

symbol $T_{[1,1,1]}$ generating a is generated, $+^{([1,1],[1,2])}_{[1]}$ is converted into $+^{([1,1,1],[1,2])}_{[1]}$. This subscript indicates that we have finished generating up to node $[1,1,1]$ in the subtree of the first child node. By storing the generated activities, we can make generation rules without discontinuity instead of increasing the number of non-terminal symbols by the type of superscript.

To obtain generation rules with no discontinuity, we need to find the *states* that represent the progress of each node under the PARALLEL operator and the *transitions* between the states. We begin with the acquisition of states.

Fig. 6 illustrates the proposed method's concept of states and transitions. In the proposed method, two states $\{v^s, v^e\}$, "start" and "end", respectively, are set at any node v in the subtree. At any node v that is not PARALLEL, $St(v)$ is defined as follows;

$$St(v) = \{v^s, v^e\} \cup \bigcup_{i=1}^{|v|} St(\mathrm{ch}(v,i)).$$

The PARALLEL operator holds the progress for each subtree of child nodes. Thus, the combination of the states of each subtree becomes the state of the PARALLEL operator as follows;

$$St(v) = \{[s_1, \ldots, s_{|v|}] | [s_1, \ldots, s_{|v|}] \in St(\mathrm{ch}(v,1)) \times \cdots \times St(\mathrm{ch}(v,|v|))\}$$

where $St(v)$ is the set of states in the descendant nodes of node v.

A transition is represented by a pair (s, v, s') of a state s, the next state s', and a state-changing node v. The set of transitions in the subtree of node v is denoted by $Tr(v)$.

For operators other than PARALLEL, the parent node's state transitions hierarchically to the child node's state. Possible transitions depend on the node operator. The arcs in Fig. 6 represent the transitions for each operator. Under SEQUENCE, any child node's end state transitions in turn to the next child node's start state. Only the last child node's end state transitions to the SEQUENCE end state. The SELECTION start state can transition to any child node's start state. All child nodes' end states transition to the SELECTION end state. Under LOOP, only the first child node's end state can transition to the LOOP end state. The last child node's end state transitions to the first child node's start state again. The ACTIVITY node's start state transitions to its end state directly. Only the transition in ACTIVITY outputs the corresponding event $l(v)$. The transitions for each operator are shown in Table IV.

TABLE IV: Transitions for each operator (except PARALLEL)

$o(v)$	$Tr(v)$
$\rightarrow$	$\{(v^s, \epsilon, \mathrm{ch}(v,1)^s)\} \cup \bigcup_{i=1}^{\lvert v\rvert-1}\{(\mathrm{ch}(v,i)^e, \epsilon, \mathrm{ch}(v,i+1)^s)\}$ $\cup\{(\mathrm{ch}(v,\lvert v\rvert)^e, \epsilon, v^e)\} \cup \bigcup_{i=1}^{\lvert v\rvert} Tr(\mathrm{ch}(v,i))$
$\circlearrowleft$	$\{(v^s, \epsilon, \mathrm{ch}(v,1)^s)\} \cup \bigcup_{i=1}^{\lvert v\rvert-1}\{(\mathrm{ch}(v,i)^e, \epsilon, \mathrm{ch}(v,i+1)^s)\}$ $\cup\{(\mathrm{ch}(v,\lvert v\rvert)^e, \epsilon, \mathrm{ch}(v,1)^s)\}$ $\cup\{(\mathrm{ch}(v,1)^e, \epsilon, v^e)\} \cup \bigcup_{i=1}^{\lvert v\rvert} Tr(\mathrm{ch}(v,i))$
$\times$	$\bigcup_{i=1}^{\lvert v\rvert}\{(v^s, \epsilon, \mathrm{ch}(v,i)^s)\} \cup \bigcup_{i=1}^{\lvert v\rvert}\{(\mathrm{ch}(v,i)^e, \epsilon, v^e)\}$ $\cup \bigcup_{i=1}^{\lvert v\rvert} Tr(\mathrm{ch}(v,i))$
T	$\{(v^s, v, v^e)\}$

In PARALLEL, any one state of the combined states transitions to the next state. For example, suppose that transition $([1,1]^s, \epsilon, [1,1,1]^s)$ is obtained from a child node (transition from the SEQUENCE node to the ACTIVITY a in Fig. 6). If there is a state $[[1,1]^s, [1,2]^s, [1,3]^e]$, then a transition $([[1,1]^s, [1,2]^s, [1,3]^e], \epsilon, [[1,1,1]^s, [1,2]^s, [1,3]^e])$ is added to $Tr(v)$ for the PARALLEL operator.

Generation rules are obtained from $Tr(v)$ for PARALLEL. The state before and after the transition correspond to the left and right sides of a generation rule, respectively. If the state of the transition is ACTIVITY, then the non-terminal symbol of the corresponding activity is generated simultaneously. That is, if $(s, v', s') \in Tr(v)$ and v' is any (ACTIVITY) node, add $+^s_v \Longrightarrow T_{v'}, +^{s'}_v$ to the generation rules. Otherwise, add $+^s_v \Longrightarrow +^{s'}_v$ to the generation rules.

C. Normalization for Removing Silent-activities

If there is silent-activity in the child nodes of the LOOP operator, the number of candidates for the trace tree can be infinite. We refer to a generation rule that generates nothing, such as $T \Longrightarrow \epsilon$, as an *$\epsilon$-rule*. Let us assume that there are two rules $(\circlearrowleft \Longrightarrow \circlearrowleft, T)$ and $(T \Longrightarrow \epsilon)$. In this case, the number of alignment candidates becomes infinite. Thus, $\Pr(\sigma; G)$ cannot be computed explicitly. Such silent-activity loops can appear in many process models because they can occur when a child node of the LOOP operator has a SELECTION operator with silent-activity (see Fig. 7 in Section VII).

In PGPM, we can apply a normalization that transforms the model into an equivalent model with the ϵ-rules removed. In CFG, the ϵ-rules can be removed by the following procedure.

- Obtain a set Φ_M of non-terminal symbols reachable in ϵ by the following procedure.
  - Add A to Φ_M if there is $A \Longrightarrow \epsilon \in R$.
  - If there is $A \Longrightarrow B \in R$ for $B \in \Phi_M$, add A to Φ_M. This is repeated until no new symbols are added.
- For all $B \in \Phi_M$, if there is $(A \Longrightarrow B, C)$ or $(A \Longrightarrow C, B)$, add $A \Longrightarrow C$ to R.

- Remove all ϵ-rules from R.

This normalization converts $A \Longrightarrow B, C$ to $A \Longrightarrow C$ if $B \Longrightarrow \epsilon$. Converted CFG is always equal to the original CFG [15].

In PGPM, the normalization is applied after the process model is converted to generation rules. These normalized generation rules ensure that the number of candidate trace trees for a trace is always finite. That is, the trace probabilities are always computable.

VI. Trace Tree Inference and Parameter Estimation

Since PGPM is a type of PCFG, the inference of trace trees and the estimation of probability parameters can utilize algorithms commonly used for PCFG.

A. Trace Tree Inference

Trace tree inference is to obtain the set of all possible trace trees $\mathbf{T}_\sigma$ and the trace probability $\Pr(L) = \sum_{\mathcal{T}_\sigma \in \mathbf{T}_\sigma} \Pr(\mathcal{T}_\sigma)$ for trace σ. For inference, we used the modified Cocke-Kasami-Younger algorithm (CKY) [14] shown in Algorithm 1. Unlike the original CKY, which remembers only the most probable tree, this algorithm remembers all possible trees. In this algorithm, the probability that subtrace $\langle w_i, \ldots, w_j \rangle$ is generated from a non-terminal symbol A is recorded in $prob_{i,j}[A]$. If $prob_{1,|\sigma|}[\$] > 0$, then the trace $L = \langle w_1, \ldots, w_{|\sigma|} \rangle$ can be generated from the model G and $Pr(L) = prob_{1,|\sigma|}[\$]$. The generation rules applied to A and the positions containing the transformed non-terminal symbols are stored in $pointer_{i,j}[A]$. Therefore, we can obtain a set of trace trees by tracing the rules and numbers contained in $pointer_{i,j}[\$]$ in order.

The key point of this algorithm is that $pointer_{i,j}$ can be obtained from the pairs of $pointer_{i,k}$ and $pointer_{k+1,j}$, $\forall k = i \ldots j$. Therefore, this algorithm first finds the non-terminal symbols $pointer_{i,i+1}$ that can generate $\langle w_i \rangle$ and their probability $prob_{i,i+1}$ (L1-4). Then, by combining the obtained $pointer_{i,i+1}$ and $pointer_{i+1,i+2}$, non-terminal symbols (and their probabilities) that can generate longer partial traces are recursively obtained (L10-18). This algorithm is a type of dynamic programming. Similar to the CKY, the computational complexity of Algorithm 1 is $O(|\sigma|^3)$.

In L5-9 and L19-23, A is also added to $pointer_{i,j}$ if there is B in $pointer_{i,j}$ and there is a rule $A \Longrightarrow B$. Note that a recursive search is also required for A at this time (L9, L23).

B. Parameter Estimation

The estimation, obtaining the probability parameter P, is difficult because actual trace trees are not always given. Existing methods estimate probability parameters from the number of transitions performed in the model by finding alignments corresponding to traces. Let $\mathrm{C}(r, \mathcal{T})$ be the frequency of rule r in the trace tree $\mathcal{T}$. If the true trace tree $\hat{\mathcal{T}}_\sigma$ for all traces σ is known, the frequency of rule r in the event log L can be calculated as $\sum_{\sigma^m \in L} m\mathrm{C}(r, \hat{\mathcal{T}}_\sigma)$. However, $\hat{\mathcal{T}}_\sigma$ is not obvious since trace trees are not always unique.

The proposed method estimates probability parameters on the basis of the expectation of rule frequency from obtained $\mathbf{T}_\sigma$. We denote the expectation of the occurrence frequency of rule r in L as ξ_r. In accordance with the probabilities of the trace trees, ξ_r is obtained as follows.

$$\xi_r = \sum_{\sigma^m \in L} \sum_{\mathcal{T} \in \mathbf{T}_\sigma} \frac{\Pr(\mathcal{T})}{\sum_{\mathcal{T}' \in \mathbf{T}_L} \Pr(\mathcal{T}')} m\mathrm{C}(r, \mathcal{T}).$$

Once ξ_r is obtained, $P(A \Longrightarrow \alpha)$ is obtained as $\xi_{A \Longrightarrow \alpha} / \sum_{r \in A \Longrightarrow \bullet} \xi_r$ where $(A \Longrightarrow \bullet)$ is the set of rules with A on the left side.

In practice, ξ_r requires $\Pr(\mathcal{T})$ to obtain ξ_r, and $P(r)$, namely ξ_r, is required to obtain $\Pr(\mathcal{T})$. In our implementation, we give some initial value to P and update ξ_r and $P(r)$ recursively. This method is a kind of EM algorithm. Therefore, repeated updates are guaranteed to converge to a probability parameter that maximizes the likelihood of the event log.

Algorithm 1 probabilisticCKY($\sigma = \langle w_1, \ldots, w_{|\sigma|} \rangle, G$)

1: **for** $i = 0, \ldots, |\sigma| - 1$ **do**
2: **for** $A \in \{X | X \Longrightarrow w_{i+1} \in R\}$ **do**
3: $prob_{i,i+1}[A] \leftarrow P(A \Longrightarrow w_i)$
4: $pointer_{i,i+1}[A] \leftarrow \{(A \Longrightarrow w_{i+1}, (i, i+1))\}$
5: **repeat**
6: **for all** $(A, B) \in \{(X, Y) | X \Longrightarrow Y \in R, X \notin pointer_{i,i+1}, Y \in pointer_{i,i+1}\}$ **do**
7: $prob_{i,i+1}[A] \leftarrow prob_{i,i+1}[A] + prob_{i,i+1}[B] \times P(A \Longrightarrow B)\}$
8: $pointer_{i,i+1}[A] \leftarrow pointer_{i,i+1}[A] \cup \{(A \Longrightarrow B, (i, i+1))\}$
9: **until** no more symbols are added to $prob_{i,i+1}$
10: **for** $l = 2, \ldots, |\sigma|$ **do**
11: **for** $i = 0, \ldots, |\sigma| - l$ **do**
12: $j \leftarrow i + l$
13: **for** $k = i + 1, \ldots, j - 1$ **do**
14: **for** $S_1 \in \{X | prob_{i,k}[X] \rangle 0\}$ **do**
15: **for** $S_2 \in \{X | prob_{k,j}[X] \rangle 0\}$ **do**
16: **for** $A \in \{X | X \Longrightarrow S_1, S_2 \in R$ **do**
17: $prob_{i,j}[A] \leftarrow prob_{i,j}[A] + P(A \Longrightarrow S_1, S_2) \times prob_{i,k}[S_1] \times prob_{k,j}[S_2]$
18: $pointer_{i,j}[A] \leftarrow pointer_{i,j}[A] \cup \{(A \Longrightarrow S_1, S_2, ((i,k),(k,j)))\}$
19: **repeat**
20: **for all** $(A, B) \in \{(X, Y) | X \Longrightarrow Y \in R, X \notin pointer_{i,j}, Y \in pointer_{i,j}\}$ **do**
21: $prob_{i,j}[A] \leftarrow prob_{i,j}[A] + prob_{i,j}[B] \times P(A \Longrightarrow B)\}$
22: $pointer_{i,j}[A] \leftarrow pointer_{i,j}[A] \cup \{(A \Longrightarrow B, (i,j))\}$
23: **until** no more symbols are added to $prob_{i,j}$
return $prob, pointer$

VII. Evaluations

In our experiments, the effectiveness of PGPM and the proposed method was verified from three perspectives; (i) Computability of trace probabilities from process models that were previously inapplicable. (ii) The goodness of fit between the inferred probabilistic process model and the event log. (iii) Accuracy and speed of conformance checking between models and traces.

In our experiments, we define *conformance checking* as the problem of binary classification of traces. That is, a trace is considered to *conform* to a process tree if the process tree can generate that trace. In PGPM, we consider σ conforms to G if $\mathrm{P}(\sigma; G) > 0$.

A. Evaluation of Inference in Models with Infinite Loops

First, we verified that PGPM infers the correct trace tree. In this experiment, we targeted the process tree shown in Fig. 7. In this model, silent-activity is present under the LOOP

operator. Thereby, the number of candidate alignments is infinite for any given conforming trace.

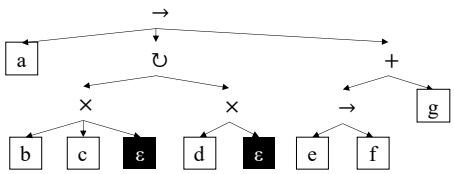


Fig. 7: Process model for experiment

The probability parameters were estimated on the basis of the event logs generated from the process tree in Fig. 7. The event log was constructed by generating 1000 traces from the process tree in this experiment. All probability distributions for each operator were assumed to be uniformly distributed.

PGPM obtained the process of generating traces and the trace probabilities correctly. As an example, the trace tree for a trace $\langle a, c, d, b, c, e, g, f \rangle$ is represented in Fig. 8. The estimated trace tree represents the generation process of the corresponding trace. The product of the probabilities of the generating rules is the probability of the trace, $Pr(\langle a, c, d, b, c, e, g, f \rangle | G) = 0.031\%$.

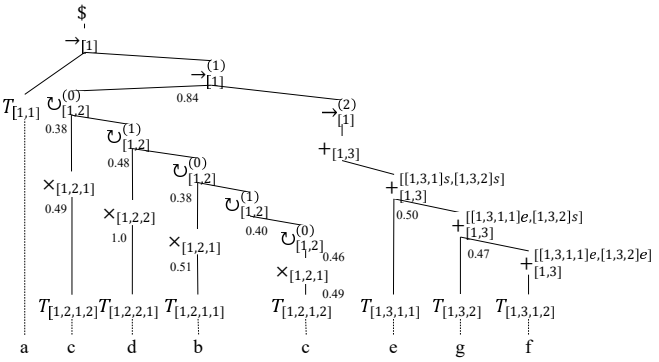


Fig. 8: Estimated trace tree

Despite the presence of silent-activity, PGPM can obtain trace probabilities. Fig. 8 shows that in the normalized PGPM, the rule that generates silent-activity $\circlearrowleft^{(1)}_{[1,2]} \Longrightarrow \times_{[1,2,2]}, \circlearrowleft^{(0)}_{[1,2]}$ is converted to a rule that does not generate activity, such as $\circlearrowleft^{(1)}_{[1,2]} \Longrightarrow \circlearrowleft^{(0)}_{[1,2]}$. PGPM enables us to construct an equivalent PCFG with the ϵ-rules removed to obtain trace probabilities.

B. Evaluation of Probability Parameter Estimation

Next, we examined the differences between existing probabilistic process model discovery methods and probabilistic parameter estimation with PGPM. We used the GDT-SPN discovery method [8] as a comparison. Similar to PGPM, GDT-SPN takes a process model (Petri Net) and event logs as input. GDT-SPN calculates probabilities for transitions in the model on the basis of optimal alignments to traces.

We used the process model shown in Fig. 9 as input to both methods. We call the probability parameters in this model (i)SINGLE, (ii)LOOP, (iii)REPEAT, and (iv)FINISH, respectively. The event log $\{\langle a \rangle^{1900}, \langle a, a \rangle^{90}, \langle a, a, a \rangle^{9}, \langle a, a, a, a \rangle^{1}\}$ was given as the input. The probabilistic process model and event log fit best when (ii)LOOP, which indicates the probability of a looping, is 50% and (iii)REPEAT, which is the probability of a being regenerated, is 10%.

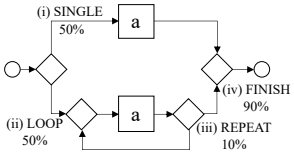


Fig. 9: Process model

TABLE V: Estimated probabilities

Probabilities (True value)	GDT-SPN	PGPM
(i)SINGLE (50%)	0.95	0.50
(ii)LOOP (50%)	0.05	0.50
(iii)FINISH (10%)	0.53	0.10
(iv)REPEAT (90%)	0.47	0.90
log-likelihood	-496.2	**-432.9**

Table V shows the probabilistic parameter estimation results for the GDT-SPN and the proposed method. Table V also shows the log-likelihood of each probabilistic process model. For GDT-SPN, the values of the corresponding probabilities are shown instead of the probabilities of the generation rules. The probabilistic process model estimated by the proposed method showed a higher log-likelihood than the GDT-SPN. That is, the PGPM highly fits with the event log.

In GDT-SPN, (i)SINGLE is much higher than the actual value of 50%. This deviation is because the probability estimation in GDT-SPN depends only on the optimal alignment. In the business process of this experiment, there are two paths to generate $\langle a \rangle$. The GDT-SPN method considered that all 1900 $\langle a \rangle$ were generated from paths that did not loop. Therefore, the probability of LOOP was low. In contrast, the proposed method estimates $\xi_{\times_{[1]} \Longrightarrow T_{[1,1]}}$ and $\xi_{\times_{[1]} \Longrightarrow \circlearrowleft_{[1,2]}}$ as 50% for each trace $\langle a \rangle$. The probability parameters were correctly estimated by considering two alignments for $\langle a \rangle$.

This model is just an obvious example of the limitation of GDT-SPN estimation. In practice, however, the GDT-SPN may not be suitable for business processes that have multiple paths for a single trace. In contrast, PGPM estimation converges to the probability parameter that maximizes the likelihood, even when there are multiple trace trees for a single trace.

C. Conformance Checking Performance with PGPM

We compared the performance of conformance checking in PGPM with those of conventional methods. For the experiments, process trees and traces were artificially generated. We randomly generated 100 process trees using the method described in [16]. This method uses the implementation in PM4Py [17], a Python library for process mining. We set $|\mathcal{E}| = 10$ and the number of nodes in the process tree to be from 5 to 15. The probability of a leaf node being ϵ is set to 10%. The four operators to be generated are shown in Table I, and the probability of occurrence of an operator is assumed to be uniform. The parameter "duplicate", which indicates the probability of multiple occurrences of activities with the same event, was tried under several conditions.

Ten conforming traces were generated using the PM4Py implementation from each process tree. We also prepared the same number of non-conforming traces generated from another process tree.

We performed conformance checking on this total of 2000 traces with the proposed and existing methods and measured the total computation time. Experiments were conducted on a

terminal with an Intel Core i5-1135G7 2.40GHz processor and 16GB RAM. We also measured the ratio of correct answers, i.e., the ratio of traces for which conformance to the process tree was correctly determined.

Token-based replay (TBR) [13] and Alignment-based replay (ABR) [9] were used as comparison methods for conformance checking. The method of Boltenhagen et al. (A∗SP) [12], an extension of ABR, was also used for comparison because it is a state-of-the-art method. In A∗SP, the discount parameter, which adjusts the calculation speed and accuracy, is set to 2 to prioritize calculation speed. All of them used implementations of PM4Py. For all methods, only the time for conformance checking for each trace was measured, and the preprocessing (model parameter estimation) time was not included.

TABLE VI: Performance measurement results

parameter	Computation Time (sec.)			
"duplicate"	ABR	A∗SP	TBR	PGPM
0.0	134.7	11.89	3.631	**3.119**
0.1	71.39	10.23	**3.286**	4.694
0.2	47.59	10.82	3.425	**3.177**
parameter	Correct Answer Ratio			
"duplicate"	ABR	A∗SP	TBR	PGPM
0.0	**1.0**	0.82	0.99	**1.0**
0.1	**1.0**	0.82	0.88	**1.0**
0.2	**1.0**	0.80	0.83	**1.0**

Table VI shows the computation time and correct answer ratio results. Both the proposed method and ABR are considered to conform only to those traces that the process tree could generate. However, ABR took more than ten times longer than the proposed method. A∗SP and TBR are faster than ABR. In particular, TBR had almost the same computation time as the proposed method. However, A∗SP and TBR made many conformance checking mistakes, especially as the DUPLICATE probability increased. These methods do not always search all activities when multiple activities correspond to an event in the trace replay. Therefore, the higher the duplicate probability, the more missed correct alignments. In contrast, despite its high speed, the proposed method searches all possible alignments. We demonstrated that the proposed method is faster and more accurate than existing methods for conformance checking.

In this experiment, only the traces generated from the process tree conformed to the converted PGPM. In other words, the PGPM and the process tree had the same language. For a process tree composed of standard operators, such as the one used in this experiment, the converted PGPMs would seem to have equal languages. However, it has not been formally proven that the language of any given process tree and converted PGPM is always equal. A mathematical proof of the equivalence of the language of the tree and the PGPM is our future work.

VIII. Conclusion

In this paper, we proposed the Probabilistic Generative Process Model (PGPM) that extends any process tree to gain trace probabilities. We also proposed a method to optimize the stochastic parameters of the model from observed data. This algorithm can efficiently determine the correspondence between traces and models and can perform conformance checking with high speed and accuracy.

PGPM allows PCC to be performed on various process models. In other words, we can use probability-based model evaluation and analysis for various process mining tasks. We expect that PGPM will accelerate the overall research in process mining.

In PGPM, traces that do not conform to the model have a trace probability of zero. In contrast, existing methods can find alignments with the fewest deviations for traces. The deviations of a trace and a model also need to be quantified for process mining. Future work may include investigating methods for quantifying the trace deviations in PGPM. We also want to prove the equivalence of the language of PGPM and the original process tree as future work.

References

[1] Leemans, J.J.S., van der Aalst, W.M.P., Brockhoff, T., Polyvyanyy, A.: Stochastic process mining: Earth movers' stochastic conformance. In Information Systems, 102(C), (2021)

[2] Leemans, J.J.S., Polyvyanyy A.: Stochastic-Aware Conformance Checking: An Entropy-Based Approach. In International Conference on Advanced Information Systems Engineering pp. 217–233. Springer, Cham (2020)

[3] van der Aalst, W.M.P., Buijs J., van Dongen, B.: Towards Improving the Representational Bias of Process Mining. In International Symposium on Data-Driven Process Discovery and Analysis, pp. 39–54. Springer, Berlin, Heidelberg (2011)

[4] van Zelst, S.J., Leemans J.J.S.: Translating Workflow Nets to Process Trees: An Algorithmic Approach. Algorithms, 13(11):279, https://doi.org/10.3390/a13110279, (2020)

[5] Molloy M. K.: Performance Analysis Using Stochastic Petri Nets, In IEEE Transactions on Computers, 31(9), pp. 913–917 (1982)

[6] Ciardo, G., German, R., Lindemann, C.: A characterization of the stochastic process underlying a stochastic Petri net. In IEEE TSE, 20(7), pp. 506–515 (1994)

[7] Cabasino, M.P., Hadjicostis, C.N., Seatzu, C.: Probabilistic marking estimation in labeled Petri nets. In 52nd IEEE Conference on Decision and Control, pp. 6304–6310 (2013)

[8] Rogge-Solti, A., van der Aalst, W.M.P., Weske, M.: Discovering stochastic Petri nets with arbitrary delay distributions from event logs. In BPM Workshops. pp. 15–27 (2013)

[9] Adriansyah, A., van Dongen B.F., van der Aalst W.M.P.: Conformance Checking using Cost-based Fitness Analysis. In 2011 ieee 15th International Enterprise Distributed Object Computing Conference, pp. 55–64. IEEE (2011)

[10] van der Aalst, W.M.P.: Decomposing Petri nets for process mining: A generic approach. Distributed and Parallel Databases, 31(4), pp. 471–507. (2013)

[11] Lee, W.L.J., Verbeek, H. M. W., Munoz-Gama, J., van der Aalst, W.M.P., Sepúlveda, M.: Recomposing conformance: Closing the circle on decomposed alignment-based conformance checking in process mining. Information Sciences, 466, pp. 55–91. (2018)

[12] Boltenhagen, M., Chatain, T., Carmona, J.: A Discounted Cost Function for Fast Alignments of Business Processes. In International Conference on Business Process Management. pp.252–269. Springer, Cham (2021)

[13] Berti, A., van der Aalst, W.M.P.: Reviving Token-based Replay: Increasing Speed While Improving Diagnostics. In ATAED@ Petri Nets/ACSD, pp.87–103. (2019)

[14] Jelinek, F., Lafferty D.J., Mercer L.R.: Basic Methods of Probabilistic Context Free Grammars. In Speech Recognition and Understanding, pp. 345–360. Springer, Berlin, Heidelberg (1992)

[15] Hopcroft, E.J., Motwani R., Ullman, D.J.: Normal Forms for Context-Free Grammars. In Introduction to Automata Theory, Languages, and Computation. 3rd edn., Addison Wesley pp.255–273. (2006)

[16] Jouck, T., Depaire, B.: PTandLogGenerator: A Generator for Artificial Event Data. In BPM (Demos), 1789, pp. 23–27. (2016)

[17] Berti, A., van Zelst, S.J., van der Aalst, W.M.P.: Process Mining for Python (PM4Py): Bridging the Gap Between Process-and Data Science. In ICPM Demo Track 2019, co-located with 1st International Conference on Process Mining, pp. 13–16. http://ceur-ws.org/Vol-2374/, Aachen, Germany (2019)