Calculating the Number of Unique Paths in a Block-Structured Process Model

Gert Janssenswillen[1,2], Benoît Depaire[1], and Toon Jouck[1]

Hasselt University, Agoralaan Bldg D, 3590 Diepenbeek, Belgium
Research Foundation Flanders (FWO), Egmontstraat 5, 1060 Brussels, Belgium
gert.janssenswillen, benoit.depaire, toon.jouck@uhasselt.be

Abstract. Estimating the number of execution paths in a process model is a non-trivial task as one runs quickly into an combinatorial explosion of possible paths. This paper introduces a new algorithm to calculate the number of different execution paths for finite-behavior block-structured models in a computationally efficient way. Block functions are defined for the workflow constructs *sequence, parallel, exclusive choice* and *finite loops*, such that the amount of behavior in each block-construct can be computed efficiently. Subsequently, the block-structuredness of the model is exploited to efficiently calculate the number of unique paths in the model. The algorithm has been implemented for process trees, although the translation to other modeling notations is straightforward. An empirical analysis showed that the run-time of the algorithm is very low, and only slightly impacted by the complexity of the model.

Keywords: Process Modeling, Process Mining, Process Trees, Process Model Complexity

1 Introduction

In the field of process mining and process modeling, there are certain situations where it is desirable to quantify the amount of behavior of a process model. For example, if one wants to quantify the *variance* of behavior present in a process model. Related to this, the *precision* measure for discovered process models, expresses to what extent the behavior in the model does not exceed the behavior in the log [3]. Similarly, the implicit realism measure [4] uses the number of unique paths in a model to calculate the probability that a certain amount of behavior from the model did not show up in the log. The amount of behavior in a model can moreover be used as a proxy for model complexity. As it can be computationally hard to compute the amount of behavior, several metrics to calculate model complexity use proxies instead. [6].

Determining the amount of behavior in a process model, which we quantify in this paper as the number of unique (execution) paths, is a challenging task. One could naively traverse the process model recursively and count the number of unique paths, but this quickly becomes computationally infeasible due to a combinatorial explosion of different (parallel) paths.

The main idea presented in this paper is to compute the number of unique paths in a block-structured finite-behavior process model in a more intelligent and computationally efficient way. As we will show, this is possible by exploiting the block-structuredness of the model. In this paper, we make the following contributions:

- We construct a block function, which calculates the number of unique paths, for each of the following process constructs: *sequence* ($\rightarrow$), *exclusive choice* ($\times$), *parallelism* ($\wedge$) and *structured finite loops* ($\circlearrowleft^k$).
- We develop a generic approach to determine the total amount of behavior in a block-structured finite-behavior process model.
- We provide an implementation for process trees.

Section 2 describes the general approach used by the algorithm, while in Section 3 the implementation is elaborated upon. The performance of the technique in terms of run-time is discussed in Section 4 while Section 5 concludes the paper.

2 General approach

In this section, the formal approach of the calculation will be described. First, some assumptions will be make regarding the types of models taken into account. In the subsequent paragraphs, different block functions for each of the specific operator type will be defined. Finally, some limitations to the formal approach will be pointed out, together with work-arounds in order to solve them.

2.1 Assumptions and used notations

It is important to keep in mind that we impose two restrictions on the process models. Firstly, we assume finite-behavior models, since it would otherwise make no sense to determine the number of unique traces. Consequently, loops in our models have a maximum number of repetitions. While this appears very restrictive, this can be justified by accepting a so-called *fairness assumption*, which states that a task of a process cannot be postponed indefinitely. This assumption therefore rules out infinite behaviors that are considered unrealistic [1]. Secondly, we assume that models are block-structured, i.e. they can be decomposed in properly nested subprocesses.

For the development and discussion of our approach, we will use the process tree notation, because process trees are block-structured by definition. However, the ideas in this paper are applicable to other notation languages as long as the models are block-structured and finite in behavior. We formally define a finite-behavior Process Tree, which is largely based on the definition in [2], as follows:

Definition 1. *Let $\mathcal{A}$ be the activity alphabet and $A \subseteq \mathcal{A}$ be a finite set of activities,then $PT = (N, r, m, c)$ is a process tree such that:*

– *N is a non-empty finite set of nodes consisting of operator (N_O) and leaf nodes (N_L) such that: $N_O \cap N_L = \emptyset$*
– *$r \in N_O$ is the root node of the tree*
– *$O = \{\rightarrow, \times, \wedge, \circlearrowleft^k, \vee\}$, the set of operator types.*
– *$m : N \rightarrow A \cup O$ is a mapping function mapping each node to an operator or activity:*

$$m(n) = \begin{cases} a \in A \cup \{\tau\}, & \text{if } n \in N_L. \\ o \in O, & \text{if } n \in N_O. \end{cases}$$

– *$c : N \rightarrow N^*$ is the direct-child-relation function:*
$c(n) = \langle\rangle$ if $n \in N_L$
$c(n) \in N^+$ if $n \in N_O$
such that

- *each node except the root node has exactly one parent:*
  $\forall n \in N \backslash \{r\} : \exists p \in N_O : n \in c(p) \wedge \nexists q \in N_O : p \neq q \wedge n \in c(q)$;
- *the root node has no parent:*
  $\nexists n \in N : r \in c(n)$;
- *each node appears only once in the list of children of its parent:*
  $\forall n \in N : \forall_{1 \leq i < j \leq |c(n)|} : c(n)_i \neq c(n)_j$;
- *a node with a loop operator type has exactly three children such that the first child is always executed first, the second child is executed maximum k times, each time followed by the first child, and finally the third child is executed once:*
  $\forall n \in N : (m(n) = \circlearrowleft^k) \Rightarrow |c(n)| = 3$.

A process tree can have five different types of operators: sequence ($\rightarrow$), parallelism ($\wedge$), exclusive-choice ($\times$), non-exclusive choice ($\vee$) and a loop ($\circlearrowleft^k$). Fig. 1 shows a process tree and illustrates how it can be decomposed into blocks. A block always consists of a root node which determines the block type. We distinguish between an activity block, a sequence block, an exclusive choice block, a parallelism block, a structured finite-behavior loop block and a non-exclusive choice block. The example in Fig. 1 consists of 8 blocks: 5 activity blocks, 1 sequence block, 1 exclusive choice block and 1 parallelism block.

2.2 Generic approach

The generic approach to determine the number of unique paths in a block-structured finite-behavior process model is a two-step approach. First, we define for each block type a block function which calculates the number of unique paths in a block. The input for these block functions are the number of unique traces x_i in each of its child-blocks. Next we can calculate the total number of unique paths through composition of the appropriate block functions.

To illustrate, take the process tree in Fig. 1 and assume block functions $F_{\rightarrow}(x_1, \ldots, x_u)$, $F_{\times}(x_1, \ldots, x_u)$ and $F_{\wedge}(x_1, \ldots, x_u)$. The total number of unique

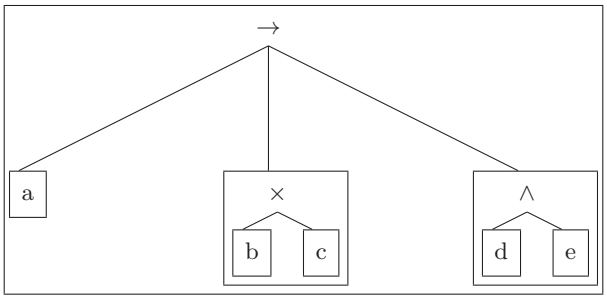


Fig. 1: Process Tree

paths in this process tree can be determined by applying the sequence block function on the outer block: $F_{\rightarrow}(x_1, x_2, x_3)$. The first block is an activity block, which implies $x_1 = 1$ as it contains only a single path. To determine the number of traces in the second and third block, we must apply the appropriate block functions again. This results in $F_{\rightarrow}(x_1, F_{\times}(x_1, x_2), F_{\wedge}(x_1, x_2))$, which can be calculated once we have defined the block functions.

2.3 Block Functions

Activity There is always only one way to execute a single activity. Therefore the activity block function is a constant value:

$$F_a = 1 \tag{1}$$

Note that silent activities, which are typically used to model specific behavior, also have $F_a = 1$ since there is in fact exactly one way to execute a silent activity.

Sequence A sequence block consists of u child-blocks such that each child-block i contains x_i unique paths. As the blocks are executed in sequence, they are executed independent from each other. Consequently, the total number of paths of a sequence block can be calculated by multiplying the number of traces in each child-block. This results in the following sequence block function:

$$F_{\rightarrow}(x_1, \ldots, x_u) = \prod_{i=1}^{u} x_i \tag{2}$$

Exclusive Choice In an exclusive choice block, only one of the different blocks will be executed at a time, therefore:

$$F_{\times}(x_1, \ldots, x_u) = \sum_{i=1}^{u} x_i \tag{3}$$

Parallelism To illustrate the development of this block function, consider the process tree in Fig. 2. Determining the number of unique paths in this tree is equivalent to determining the number of unique words that can be formed by the set of activity letters {a, b, c, d}, given specific constraints which make some words, such as "bacd", invalid.

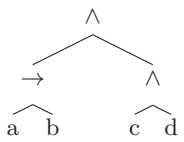


Fig. 2: A Process Tree with parallelism

To solve this challenge it is important to realize the following. Originally we have a problem of determining all 4-letter words with the letters {a, b, c, d}, given specific constraints. But once we have determined the positions of the letters of the first element, i.e. {a, b}, our problem reduces to creating a 2-letter word with the remaining letters {c, d}. This holds because any insertion of a and b in the four letter words, leaves 2 positions for the remaining letters **and** since the elements are in parallel, they impose no further restrictions on each other.

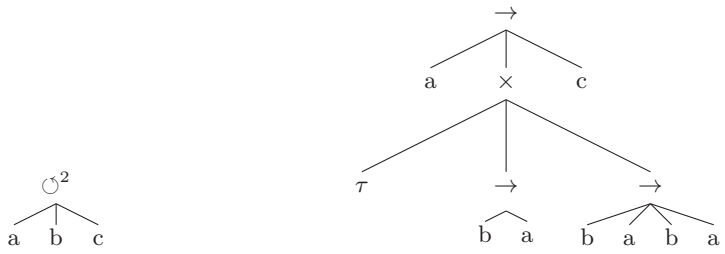
Therefore, we can tackle this problem by determining in how many ways we can validly insert the letters of the first element into a four-letter word and multiply this by the number of ways we can validly insert the letters of the second element in a two-letter word. To determine the number of ways we can insert the letters of the first and second element in respectively a four- and two-letter word is solved in two steps. First we determine the number of ways we can select the appropriate number of positions in our n-letter word and next we determine the number of ways we can validly insert our activities.

For the first element, we start by determining the number of ways we can select two positions from our four-letter word, which equals to $\binom{4}{2} = 6$ combinations and multiply this by the valid number of ways we can insert {a, b} into these two positions. The latter equals the number of paths the first child-block contains, which is 1 for a sequence of two activities. Therefore the number of ways we can validly insert {a, b} in our four-letter word equals $1\binom{4}{2} = 6$. We can repeat this for the second element, which results in $2\binom{2}{2} = 2$ ways to validly insert {c, d} in a two-letter word, since there is only one way to select 2 positions from a two letter word and there are two traces possible by a parallel construct of two activities. To conclude, the process tree in Fig. 2 has 12 unique paths.

To formalize this approach, we need some additional notation. Assume z_i to be the number of non-silent activities in child-block i, in addition to the use of x_i and u to determine respectively the number of unique paths in child-block element i and the number of child-blocks. We can then express the parallelism block function as follows:

$$F_{\wedge}(x_1, \ldots, x_u, z_1, \ldots, z_u) = \prod_{i=1}^{u} x_i \binom{\sum_{j=i}^{u} z_j}{z_i} \tag{4}$$

Structured Finite Loops A structured finite loop block is a special kind of process construct in the sense that it always contains three child-blocks [1]. The first child-block is always executed, the second child-block is executed a limited number of times (k), each time followed by the first child-block, and finally the third child-block is executed to conclude. This structure allows us to transform a finite loop into an equivalent structure using $\rightarrow$ and $\times$ nodes, as illustrated by Fig. 3



(a) Standard representation (b) Equivalent representation

Fig. 3: Finite Loop Construct

Based on the block functions $F_{\rightarrow}$ and $F_{\times}$, we can now easily see that the finite loop block function can be expressed as follows, where k represents the maximum number of loop-iterations:

$$F_{\circlearrowleft}(x_1, x_2, x_3, k) = x_1 \cdot \sum_{i=0}^{k} (x_1 x_2)^k \cdot x_3 \tag{5}$$

2.4 Limitations

Our suggested approach holds two limitations the reader should be aware of. Firstly, there is no block function for a non-exclusive choice construct. Secondly, the parallelism block function assumes that the number of activities z_i within a child-block i is fixed. However if a block contains an (exclusive) choice construct or a finite loop construct, this assumption is violated.

[1] This is so for the process tree notation. One could argue whether the third element is in fact part of the loop when considering other notations, but the point remains that it is always possible to transform a structured finite loop to a three-block construct.

Both limitations can be circumvented by preprocessing the process tree. As for the first limitation, non-exclusive choice constructs can be transformed into an exclusive choice between all possible combinations of the non-exclusive choice construct put in parallel. This is illustrated in Fig 4.

(a) Non-exclusive Choice construct (b) Exclusive Choice construct

Fig. 4: Transformation of non-exclusive choice construct.

As for the second limitation, we can always transform finite loop constructs (cf. Fig. 3) and non-exclusive choice constructs (cf. Fig. 4), such that we only have sequence, exclusive choice and parallelism constructs left. Subsequently, we can transform the tree by duplicating parts of the tree such that exclusive choice constructs only appear as parent and never as child of parallelism constructs. This transformation is illustrated in Fig. 5

(a) × as child of ∧ (b) × as parent of ∧

Fig. 5: Transformation of process tree to remove × as siblings of ∧.

Note that these are naive approaches to deal with the given limitations which should always work. However, a more efficient work-around to deal with these limitations which does not require explicit transformation of the process tree is possible, as we will show in the implementation.

3 Implementation

In our implementation, we follow a slightly different approach than suggested above such that we do not need to transform the process tree. Instead of computing the number of unique paths for each block, we compute a block dictionary for each block such that the keys represent a specific path-length (i.e. the number

of visible activities) and the values represent the number of unique paths of that specific length in the block. We define this block dictionary as

$$T = \{(z_i, x_i) | \forall (z_i, x_i), (z_j, x_j) : z_i = z_j \Rightarrow (z_i, x_i) = (z_j, x_j)\} \tag{6}$$

For example, a block with $T = \{(1, 3), (3, 2)\}$ contains 3 paths of length 1 and 2 paths of length 3. To retrieve the number of unique paths in a process tree, one has to sum over all values of the block dictionary for the root block: $\sum_{i=1}^{u} x_i$. The implementation has been put available as an R-package on *github.com/gertjanssenswillen/ptR*.

3.1 Preliminaries

Firstly, we define a function f_Z which returns the set of all paths lengths in a specific block dictionary.

$$f_Z(T) = \{z | \exists (z_j, x_j) \in T : z_j = z\} \tag{7}$$

Next, we define the function $f_X : T \times \mathbb{N} \to \mathbb{N}$, which determines how often a path of length z occurs in the block corresponding to block function.

$$f_X(T, z) = \begin{cases} 0, & \text{if } z \notin f_Z(T) \\ x, & \text{else such that } (z, x) \in T \end{cases} \tag{8}$$

Finally, we define the operator $\biguplus$ for two block dictionaries T_i and T_j as follows:

$$T_i \biguplus T_j = \{(z, x) | z \in f_Z(T_i) \cup f_Z(T_j), x = f_X(T_i, z) + f_X(T_j, z)\} \tag{9}$$

3.2 Algorithm

Algorithm 1 shows the main code of the implementation, which implements the general idea of our approach, i.e. we exploit the block-structuredness of the model. We start with the block defined by the root-node and calculate the block dictionary based on the block-type and the block dictionaries of its child-blocks. If the root-block is a visible or silent activity, its block dictionary is respectively $\{(1, 1)\}$ and $\{(0, 1)\}$ (cf. line 6-9).

In all other cases, we first determine the block dictionaries of the child-blocks (line 10-14) by applying the algorithm recursively. Next, we apply the appropriate block function based on the block type (line 15-25). These block functions are an extension of the block functions described above, since they need to compute block dictionaries instead of scalar values representing the number of paths. In the next section, we will illustrate each extended block function by means of the process tree shown in Fig. 6. Note that this process tree is annotated, i.e. each node contains a superscript identifying the node number as well as its block dictionary.

Algorithm 1 : NumberOfPaths

1: **Input:**
2: $PT = (N, r, m, c)$: A Process Tree
3: k: A maximum number of iterations for loops
4: **Output:**
5: T: a dictionary characterizing the paths in PT
6: **if** $r \in \mathcal{A}$ **then**
7: $T = (1, 1)$ ▷Tree contains one path of length one
8: **else if** $r = \tau$ **then**
9: $T = (0, 1)$ ▷Tree contains one empty path
10: **else**
11: $u = |c(r)|$
12: **for** each child c_i of PT **do**
13: $T_i = NumberOfPaths(c_i)$ ▷Call the function recursively on each of the subtrees
14: **end for**
15: **if** $r = sequence$ **then** ▷Use results and type to calculate end result
16: $T = Sequence(T_1, ..., T_u)$
17: **else if** $r = choice$ **then**
18: $T = Choice(T_1, ..., T_u)$
19: **else if** $r = parallel$ **then**
20: $T = Parallel(T_1, ..., T_u)$
21: **else if** $r = loop$ **then**
22: $T = Loop(T_1, ..., T_3, k)$
23: **else**
24: $T = Or(T_1, ..., T_u)$ ▷i.e. non-exclusive choice
25: **end if**
26: **end if**

3.3 Extended Block Functions

Sequence To illustrate the implementation of the extended sequence block function (Alg. 2), consider $\rightarrow_4$ in Fig. 6. This sequence block has two children, with the following dictionaries $\{(2, 4), (4, 24)\}$ and $\{(1, 1)\}$. We have to combine every key-value pair from the first dictionary with every key-value pair from the second dictionary (line 8-9). For any combination we have to create a new key-value pair and add it to the parent's block dictionary (line 10-12). Thus, $(2, 4)$ with $(1, 1)$ produces $(3, 4)$ and $(4, 24)$ with $(1, 1)$ results in $(5, 24)$. Note that line 10 corresponds to the general block function described before.

Parallelism To illustrate the implementation of the extended parallelism block function (Alg. 3), consider $\wedge_8$ in Fig. 6. This parallelism block has two children, with the following dictionaries $\{(1, 1\}$ and $\{(1, 1)\}$. Since both dictionaries have only one key-value pair, we only have to combine those two key-value pairs (line 8-9). To compute the key-value pair for the parent's block dictionary we apply the formulas in line 10 and 11. Thus, $(1, 1)$ and $(1, 1)$ result in $(1+1, 1\binom{2}{1}1\binom{1}{1}) =$

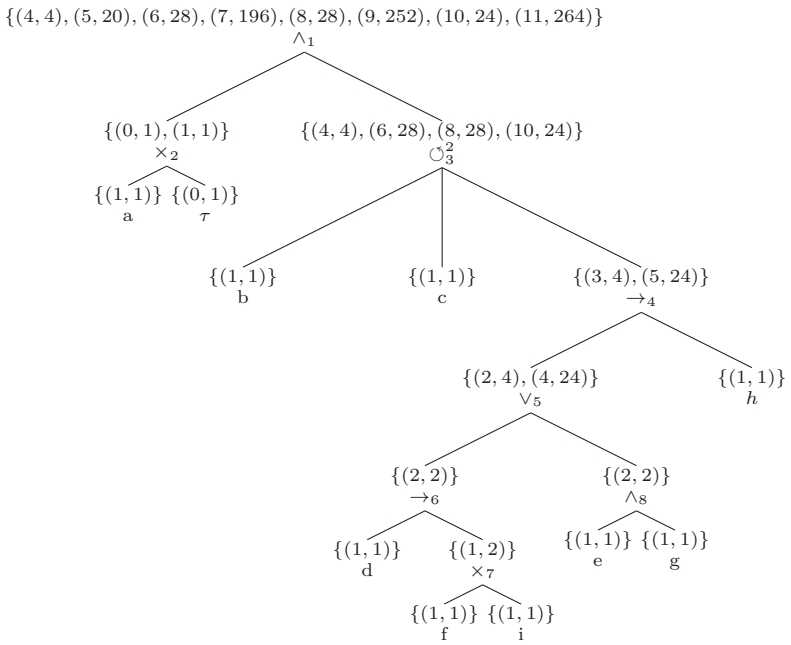


Fig. 6: Process Tree annotated with block dictionaries.

Algorithm 2 : Sequence

Input:
    $\{T_i \mid i = 1, ..., u\}$: u dictionaries representing paths in child-blocks of a sequence node
Output:
    T: a dictionary representing the paths in a sequence node
$S = T_1$
for $T_i \in T_2, \ldots, T_u$ **do**
    $R = \{\}$
    for $(z_r, x_r) \in S$ **do**
        for $(z_i, x_i) \in T_i$ **do**
            $x_0 = x_r \cdot x_i$
            $z_0 = z_r + z_i$
            $R = R \uplus \{(z_0, x_0)\}$
        end for
    end for
    S = R
end for

$(2, 2)$. Note again that line 10 corresponds to the general block function described before.

Algorithm 3 : Parallel

1: **Input:**
2: $\{T_i | i = 1, ..., u\}$: u dictionaries representing paths in children of a parallel node
3: **Output:**
4: T: a dictionary representing the paths in a parallel node
5: $R = T_1$
6: **for** $T_i \in T_2, \ldots, T_u$ **do**
7: $S = \{\}$
8: **for** $(z_r, x_r) \in R$ **do**
9: **for** $(z_i, x_i) \in T_i$ **do**
10: $x_0 = x_r \cdot \binom{z_r + z_i}{z_r} \cdot x_i \cdot \binom{z_i}{z_i}$
11: $z_0 = z_r + z_i$
12: $S = S \biguplus \{(z_0, x_0)\}$
13: **end for**
14: **end for**
15: $R = S$
16: **end for**

Exclusive Choice To illustrate the implementation of the extended exclusive choice block function (Alg. 4), consider $\times_7$ in Fig. 6. This exclusive choice block has two children, with the following dictionaries $\{(1, 1\}$ and $\{(1, 1)\}$. The exclusive choice block is a bit simpler as it just adds all the key-value pairs of the children's dictionaries to the parent block dictionary. Thus after executing lines 6-10, we have the following set of key-value pairs: $\{(1, 2)\}$. Note that for this extended block function, it is in fact the $\biguplus$ operator which corresponds to the general block function described above.

Algorithm 4 : Exclusive Choice

1: **Input:**
2: $\{T_i \mid i = 1, ..., u\}$: u dictionaries representing paths in children of a choice node
3: **Output:**
4: T: a dictionary representing the paths in a choice node
5: $R = T_1$
6: **for** $T_i \in T_2, \ldots, T_u$ **do**
7: **for** $(z_i, x_i) \in T_i$ **do**
8: $R = R \biguplus \{(z_i, x_i\}$
9: **end for**
10: **end for**

Finite Structured Loop For the finite structured loop it is not possible to apply the general block function due to our approach which requires to calculate the number of unique paths for each path size separately. Therefore we fall back to the insight, illustrated in Fig. 3, that a finite structured loop can be transformed into an equivalent structure of sequence constructs and a exclusive choice construct. To illustrate consider $\circlearrowleft_3^2$ in Fig. 6.

At lines 9-14, the code in Alg. 5 first determines the block dictionary of the exclusive choice in the transformation (cf. Fig. 3b). At first, $XORset = \{(0,1)\}$, which represents the invisible task. Next, a single repeat-block is added, which consists of a sequence of the redo and do parts. In our example, this results in $XORset = \{(0,1),(2,1)\}$. Since, the maximum iterations of the repeat-block is two, we add another block which repeats the repeat-block twice, resulting in $XORset = \{(0,1),(2,1),(4,1)\}$. Finally, we calculate the block dictionary of the entire loop-block, by applying the sequence block function to the do-block, the XOR-block and the exit-block. First it combines the do and XOR-block, which results in $\{(1,1),(3,1),(5,1)\}$. Next, it combines this with the exit block, which results in $\{(4,4),(6,28),(8,28),(10,24)\}$.

Algorithm 5 : Loop

```
1: Input:
2:       {T_1, T_2, T_3}: 3 dictionaries representing paths in the children of a loop node
3:       k: A maximum number of iterations for loops
4: Output:
5:       T: a dictionary representing the paths in a loop node
6: do = T_1
7: redo = T_2
8: exit = T_3
9: repeat = {(0,1)}
10: XORset = repeat
11: for i in 1,...,k do
12:     repeat = Sequence(repeat, redo, do)
13:     XORset = ExclusiveChoice(XORset, repeat)
14: end for
15: T = Sequence(do, XORset, exit)
```

Non-exclusive choice For the extended non-exclusive choice block function, we exploit the insight that a non-exclusive choice construct can be rewritten as an exclusive choice of parallelism construct, as illustrated in Fig. 4. This can be seen in the code in Alg. 6 on lines 6-7. Here, we iterate over all possible subsets of children, i.e. $\mathbb{P}(\{T_i\})$[2], except the empty set. To illustrate, consider $\vee_5$, which has two children with block dictionaries $\{(2,2)\}$ and $\{(2,2)\}$. When executing this choice block, one can either execute only the first child, only the second

[2] $\mathbb{P}(\{T_i\})$ refers to the set of all subsets of $\{T_1, ..., T_u\}$

child or both children. When executing only a single child, the resulting block dictionary will be that of the child. When executing both children in parallel, the block dictionary will be $\{(4, 2\binom{4}{2}2\binom{2}{2})\} = \{(4, 24)\}$. Next, the union is taken of the three block dictionaries, which results in the set $\{(2, 4), (4, 24)\}$.

Algorithm 6 : Or (non-exclusive choice)

Input:
    $\{T_i | i = 1, ..., u\}$: u dictionaries representing paths in children of an or node
Output:
    T: a dictionary representing the paths in a or node
$R = \emptyset$
for $S \in \mathbb{P}(\{T_i\}) \setminus \emptyset$ **do** ▷Iterate over all non-empty subsets of the branches
    $R = R \biguplus Parallel(S)$ ▷Calculate the paths using the Parallel function
end for

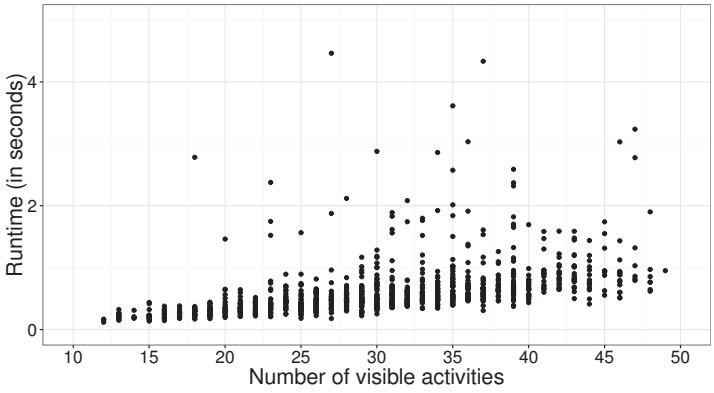
4 Performance of the algorithm

The performance of the algorithm was empirically investigated on a large collection of process trees. The trees were generated using the framework described in [5]. Each of the five constructs was given an equal probability of occurrence, i.e. 20%. The occurrence of silent transitions was set at 10%. The number of visible activities in the trees follows a triangular distribution with minimum 10, maximum 50 and a mode of 30. All experiments were executed on a workstation with 2 processors (2.30Ghz; 4 virtual threads) and 8GB of memory.

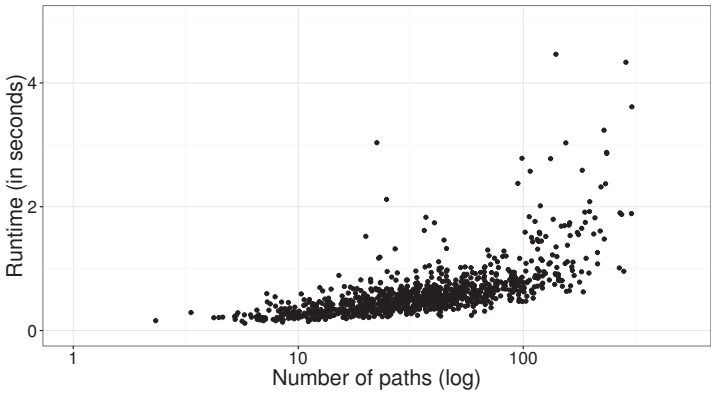
Fig. 7 shows both the run-time (in seconds) and the number of paths in relation to the number of visible activities in the tree. In Fig. 7a, it can be seen that there appears to be a linear relation between the number of visible activities and the run-time. However, the number of activities on itself is not a very precise proxy for the complexity of a tree, since the real impact stems from the operators and their relative positions in the tree. Therefore, the number of paths itself appears to be a more reliable estimate for the complexity of the tree. Fig. 7b shows the relation between the number of paths (with logarithmic transformation), as a proxy for the complexity of the tree, and the run-time. Note that due to the logarithmic transformation, the relation is actually linear.

In order to quantify the relationship between complexity, as measured by the number of paths, and run-time of the algorithm, several linear regression models were fitted on the data. As well a linear-linear model as a linear-log model, and a log-log model was composed. These results showed that the log-log model fitted the data best. The result of this regression are shown in Table 1.

The interpretation of the regression is that when the complexity increases with one order of magnitude (i.e. an increase of 1000%), the run-time will increase with $10^0.004$, or 0.8%. Thus, although a positive relation exist, it can be stated that it is almost negligible.



(a) Run-time in relation to the number of activities



(b) Runtime in relation to number of paths

Fig. 7: Influence of the number of activities and number of paths on run-time

5 Conclusions and future work

Estimating the number of execution paths in a process models is a non-trivial task. Approaches which enumerate all possible paths or traverse the state space of the model become quickly unfeasible, due to the explosion of possible paths in the presence of parallel constructs. This paper therefore introduced a new technique to calculate the number of execution paths for finite block-structured models. The technique has been implemented for process trees, but can easily be transferred to other model notations.

Instead of enumerating all the paths, the technique constructs so-called block dictionaries for each block in the process model, which contain the number of paths per given length. The result of the algorithm is an annotated process tree, where each of the operator nodes has been allocated a block dictionary describing

Table 1: Log-log regression between number of paths and runtime

	Dependent variable:
	log(runtime)
log(numberOfTraces)	0.004^{***}
	(0.0001)
Constant	-0.453^{***}
	(0.008)
Observations	985
R^2	0.490
Adjusted R^2	0.489
Residual Std. Error	0.169 (df = 983)
F Statistic	943.526^{***} (df = 1; 983)
Note:	*p<0.1; **p<0.05; ***p<0.01

the number of execution paths it contains. The number of paths in the tree can then be obtained by summing over the block dictionary of the root node.

The evaluation of the performance of the algorithm showed that even for trees with more than 10^{500} different paths, the run-time does not exceed 5 seconds. Using linear regressions, only a negligible effect of the complexity of the model on the run-time was found.

As future work, the theoretical complexity of the algorithm should be investigated, as well as a formal proof of it's completeness. Also extensions towards the ability to cope with long-term dependencies could be investigated. Furthermore, the obtained annotated process tree is expected to provide useful new insights in process complexity, as it is able to point out at which locations in the process the amount of behavior increases drastically.

References

1. Baier, C., Katoen, J.P., et al.: Principles of model checking, vol. 26202649. MIT press Cambridge (2008)
2. Buijs, J.C.A.M.: Flexible Evolutionary Algorithms for Mining Structured Process Models. Ph.D. thesis, Technische Universiteit Eindhoven, Eindhoven (2014)
3. Buijs, J.C., van Dongen, B.F., van der Aalst, W.M.P.: On the role of fitness, precision, generalization and simplicity in process discovery. In: On the Move to Meaningful Internet Systems: OTM 2012, pp. 305–322. Springer (2012)
4. Depaire, B.: Process model realism: Measuring implicit realism. In: Business Process Management Workshops. pp. 342–352. Springer (2014)
5. Jouck, T., Depaire, B.: Generating Artificial Data for Empirical Analysis of Process Discovery Algorithms: a Process Tree and Log Generator. Technical Report, Universiteit Hasselt, Universiteit Hasselt (Mar 2016)
6. Mendling, J.: Detection and prediction of errors in EPC business process models. Ph.D. thesis, Wirtschaftsuniversität Wien Vienna (2007)