UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Software Engineering Curriculum

Bogdan Semiletko

Dealing with Complex Parallel Structures in Process Discovery

Master's Thesis (30 ECTS)

Supervisor(s):
Fabrizio Maria Maggi

Tartu 2015

Dealing with Complex Parallel structures in process discovery

Abstract: One of the aims of process mining is to discover a process model from a log. However, the quality of the discovered model depends on the completeness of the information about the process behaviour contained in the log. Incomplete logs do not provide all the possible behaviours. Existing process discovery algorithms dealing with incomplete logs, have troubles when working with complex parallel structures, because parallel behaviour has factorial rate of growth with respect to the number of branches. In this work, a new algorithm is proposed, which combines divide and conquer approach, with the existing mining algorithms to improve discovery of highly structured and highly concurrent process models from incomplete logs. This work describes the proposed algorithm, and explains how it works with illustrative step-by-step examples of the mining procedure. Finally, we evaluate the effectiveness and efficiency of our approach by using process models containing complex parallel structures and randomly generated models.

Keywords: process model discovery, process mining, automated process discovery, BPMN, process trees, parallel block mining, parallel branch discovery, incomplete logs mining, log incompleteness

Keerukate paralleelstruktuuridega toimetulek protsessiavastuses

Lühikokkuvõte: Üks protsessikaeve eesmärkidest on leida protsessimudeleid logifailidest. Samas sõltub leitava protsessimudeli kvaliteet sellest, kui täielik informatsioon protsessi käitumise kohta logifailis on, kuna paralleelarvutuste keerukuse kasv on faktoraalses sõltuvuses harude hulgast. Selles lõputöös tutvustatakse uut algoritmi, mis kombineerib jaga-ja-valitse võtet olemasolevate kaevealgoritmidega, et täiustada hästistruktureeritud ja samaaegselt toimuvate tegumitega protsessimudelite kaevet poolikutest logifailidest. See töö kirjeldab väljapakutud algoritmi ja selgitab, kuidas see töötab samm-sammu haaval illustratiivsete kaeveprotsessi näidete abil. Lõpuks hindame selle meetodi efektiivsust ja tulemuslikkust kasutades protsessimudeleid, mis sisaldavad samaaegselt toimuvaid tegumeid ja juhuslikult loodud mudeleid.

Võtmesõnad: protsessikaeve, automatiseeritud protsessiavastus, BPMN

Contents

1 Introduction

In this chapter we introduce the problem to be addressed in this thesis and the research questions we want to answer. Then we sketch the contribution of the thesis aimed at answering the research questions.

1.1 Problem

Process mining is a relatively young discipline for analysing process data, and improve business processes ([1]). Process mining includes three main branches: automated process discovery, conformance checking and process enhancement. Automated process discovery aims at building process models from events logs without any apriori information. Conformance checking techniques allow users to detect discrepancies between a real behaviour of a business process, as recorded in an event log, and some expected behaviour described in a process model. Process enhancement allows users to enrich an input process model with information retrieved from logs.

In this thesis we focus on process discovery. The problem we try to address in this context, is that existing approaches fail in the discovery of process models containing complex parallel structures. This is especially true in case of input logs with high level of incompleteness, and infrequent behaviour.

Existing process discovery algorithms dealing with incomplete logs, have troubles when working with complex parallel structures, because parallel behaviour has factorial rate of growth with respect to the number of branches. Thus the existing algorithms output a model with a behaviour which does not correspond to the original process model.

In this thesis, we want to answer the following research questions.

Main research question:

How to effectively discover business process models containing complex parallel structures?

Partial research questions:

- How to discover block structured from logs with high degree of incompleteness?
- What is the best way to identify a group of events belonging to a parallel block using the information contained in an event log?
- How to cluster events belonging to a parallel block into different branches?

Research objectives:

- Implement the approach as a plug-in of the ProM platform.
- Evaluate the effectiveness of the proposed algorithm with respect to state of the art techniques.
- Evaluate the efficiency of the proposed algorithm with respect of the size of the input log.

1.2 Contribution

The proposed algorithm combines a divide and conquer approach with existing mining algorithms to improve the discovery of highly structured and highly concurrent process models from incomplete logs. We introduce two theorems, which allow us to discover groups of events belonging to parallel blocks, and to determine the events belonging to each branch in each block. To discover a correct process structure, we iteratively replace every branch of every

parallel block with a placeholder. To reconstruct a final process model, we recursively apply the algorithm using a bottom-up approach.

We implemented our algorithm as a plugin[1] of the process mining tool ProM.

To evaluate the efficiency and effectiveness of our approach, we performed several qualitative and quantitative tests. The efficiency was measured using logs with different characteristics, randomly generated starting from different process models. The effectiveness was evaluated using as a baseline the Inductive Miner. For this purpose, we generated a set of logs using the same models and characterised by different degrees of completeness.

[1] https://github.com/sjbog/PBMiner

2 Background and related work

In this chapter, prerequisites and prior information needed to understand the thesis is described.

2.1 Process models

Process models try to capture and describe the behaviour of some processes, using elements with deterministic order of execution. There exist numerous model representation formats, but we will focus on the standard solutions in the field of Business Process Management [2].

The goal of any process model is to describe order of execution of activities. Almost any execution order could be achieved with 3 base flows: sequential, parallel and exclusive choice.

Petri Net is a process modelling notation based on transition system, with 2 model elements: states (places) and transitions. It is represented as a directed graph, where transitions and states are represented by arcs and nodes respectively, and the notion of tokens to indicate the control flow. The process starts in one of the initial states and terminates in one of the final states, and a path of the graph, from start to a final state, corresponds to the process execution path. Petri Nets, as well as their sub-class Workflow Nets ([3]), are one of the most widely used model representation formats.

Business Process Modelling Notation (BPMN) is another widely used model representations standard, which was designed to describe business processes management ([2], [4]). BPMN structural elements are divided into 4 categories:

- Flow objects (events, activities and gateways)
- Connecting objects (message flow, sequence flow, etc.)
- Swim lanes (pool, lane)
- Artefacts objects (data object, annotation, etc.)

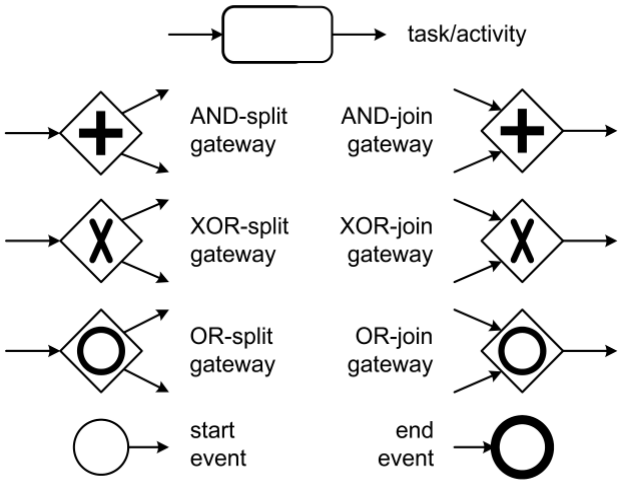

Figure 1: BPMN notation [2]

Figure 1 shows main BPMN flow elements. The gateways indicate order of execution for activities:

- AND-gateway specifies parallel behaviour, i.e., concurrent execution of all choices, without explicitly specifying the order of execution.
- XOR-gateway denotes exclusive choice behaviour, i.e., only 1 of the choices is executed.

- OR-gateway specifies non-exclusive choice behaviour, i.e., 1 or more choices is executed is any possible order.

Neither Petri Nets nor BPMN guarantees sound models, e.g. models without deadlocks or other anomalies [5], however for the implementation it is important.

Process Tree [6] is an abstract hierarchical process model representation structure that are guaranteed to be sound. Process trees are one of the most popular formats for computer software, which allow conversion from and to different standardised formats (BPMN, Petri Net, etc.). Process Tree could be described as Directed Acyclic Graph, where edges denote hierarchical relationship of respective nodes. Nodes, which are generalisation for blocks (operators) and tasks (activities), describe causal relationship (order of execution) of its children. The most used blocks have the following notation:

- `Seq()` – sequence flow of execution
- `And()` – parallel / concurrent flow of execution
- `Xor()` – exclusive choice execution

Any block could have nested blocks, however leaf nodes (tasks) are final. Children of parallel and exclusive-choice Xor blocks are often called branches.

2.2 Log representation

Data generated from the execution of process models are called event logs. Events in a log refer to actions of a model, and sometimes events are also called actions. An event log contains a set of events grouped by execution instances, called traces. All these structural elements contain a set of defined attributes, among standardized ones are:

- Id - provides unique identifiers (UUIDs) for elements.
- Timestamp – date and time, at which the element has occurred
- Name – non-unique, human understood label of element. For logs, it could be the name of the executed process.
- Lifecycle transition – specifies stage of event's execution lifecycle (usually atomic, thus complete), for example: start, schedule, suspend, resume, etc.
- Resource / role – name or identifier of the resource / role, which have triggered the event.

In addition there could be unlimited number of additional attributes. For example, costs is one of commonly used attributes, which describes action related costs, usually has 2 embedded attributes – currency and amount. Other examples would be costs, system being used and data of an instance, which allows to correlate and find causal relationships of event logs.

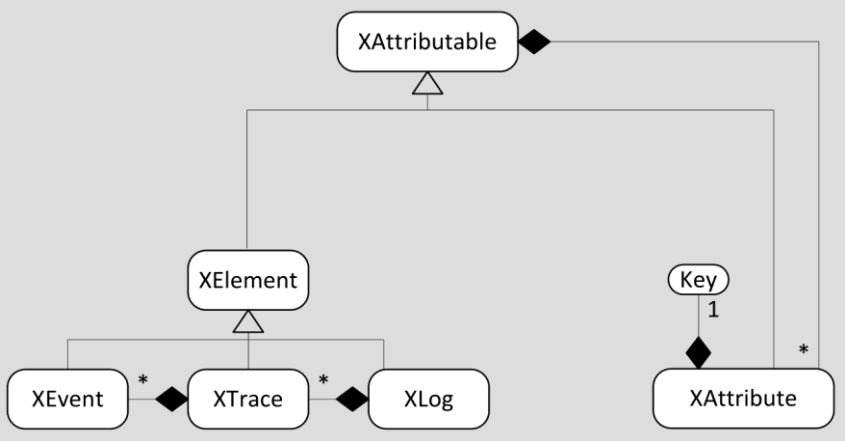

Figure 2 Structural elements of process mining event logs [7]

Usually traces in a log are unordered, whereas an events list of a trace is ordered [7]. Thus any two traces with the same order of events are usually considered equal.

There are 2 commonly used formats for log representation within BPM environment: MXML and XES standards. Although different vendors and system define numerous log formats, standardized representation allows to leverage log analysis and process mining tools like ProM.

MXML (Mining eXtensible Markup Language) is an extensible, XML-based format for storing process event logs, which emerged in 2003 and was adopted by the process mining community (ProM tools) as standard format. Since 2010, when IEEE Task Force on Process Mining adopted a less restrictive and truly extendible successor, it is considered a legacy standard [8].

XES (eXtensible Event Stream) is an open XML-based standard for storing and managing event logs [9], sometimes called OpenXES by its open-source reference implementation library [7]. It was designed primarily for process mining, with a main purpose to provide generally-acknowledged format for event logs interchange. In addition, creators made it suitable for data mining and statistical analysis.

The following principles were kept in mind, while developing XES standard:

- Simplicity – represent information in a simplest possible way, while still being human-readable, allow fast and easy log generation and parsing.
- Flexibility – aims to be general standard for event log data, without specificity or background of process mining or business processes.
- Extensibility – ability to be transparently extended in the future, while maintaining backward and forward compatibility. In addition, ability to extend with special requirements or schemas to work with specific application domain or software implementations.
- Expressivity – allow to attach human-interpretable semantics to strictly typed information elements, while aiming for generic format with as little loss of information as possible.

Since XES aims to be generic log format, only most common elements, identifiable by any setting are explicitly defined by the standard. All the other information, in particular process mining specific, is deferred to the optional attributes, and the semantics is standardized by external implementation extensions.

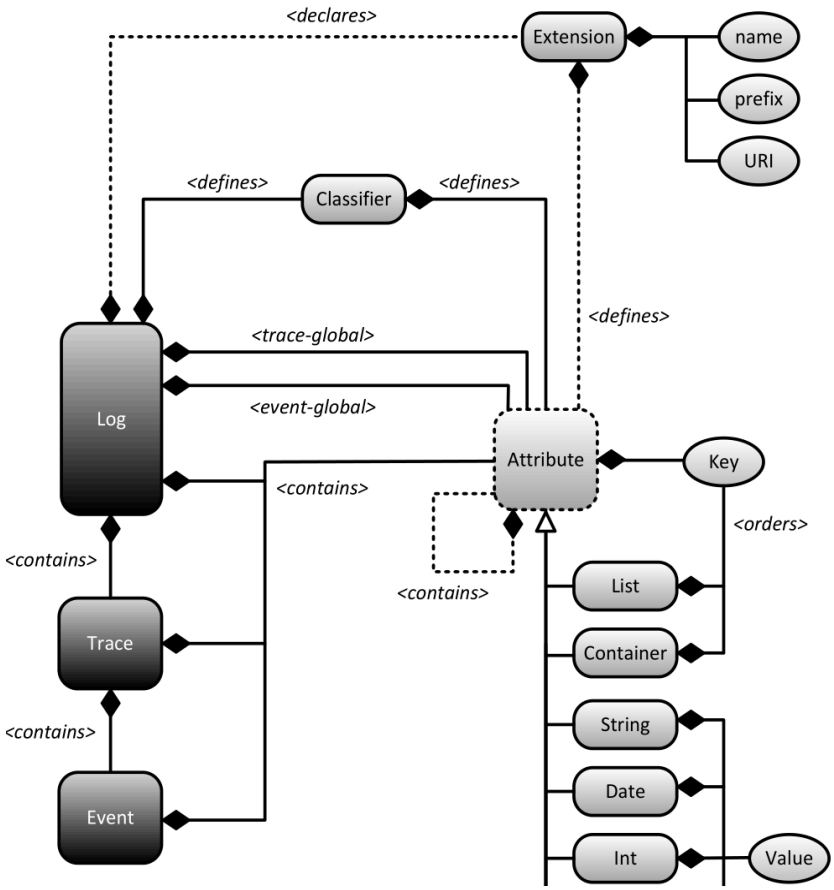

Figure 3: XES meta-model structure

The top level object is a log entity, which holds information related to the specific process, and contains traces that describe specific instance. Event objects represent atomic activities, observed during process execution. These objects don't contain the actual information, which is stored in attributes, but only define the document structure. They could contain an arbitrary number of attributes.

Attributes describe their enclosing container with a key-value pair, where keys should be unique within parent element. Standard defines commonly used attribute types: string, date, numeric, boolean, container, etc. High flexibility of standard even allows to have nested attributes.

XES uses a concept of event classifiers, which assigns an identity to each event, thus making them comparable. Classifiers are defined as a set of attributes within log's global attributes (which they are subset of), thus allowing to obtain high-level aggregate information and generate log summaries. Since XES doesn't define specific set of attributes, extensions are used to introduce a set of common attributes within a specific perspective or dimension. In addition, extensions allow to resolve ambiguity of attribute naming via attribute keys prefixing.

2.3 Existing automated process discovery algorithms

Alpha-Algorithm process discover algorithm ([3], [2]) is one the simplest and one the oldest process discovery algorithms, which could deal with concurrency. It uses straightforward naive approach to scan the log for predefined patterns and build N^2 footprint matrix. Then builds a process model, adding one activity after another, and optimises (or reduces) the discovered relations. Hence, it has a lot of limitations, for instance it cannot deal with noisy logs, infrequent

and incomplete behaviour, complex nested structures, etc. Alpha-Algorithm is often considered a baseline and often embedded into other complex process discovery algorithms.

Inductive Miner [10] is one of the best process discovery algorithms, which outputs a sound and fit block-structured model in a finite time. It applies divide and conquer strategy, first by partitioning the activities and ordering them according to predefined process construct ranking. Then, it uses most important construct as a cut point, to perform a log split. The steps are recursively repeated until base case is encountered. See example illustrated in Figure 4.

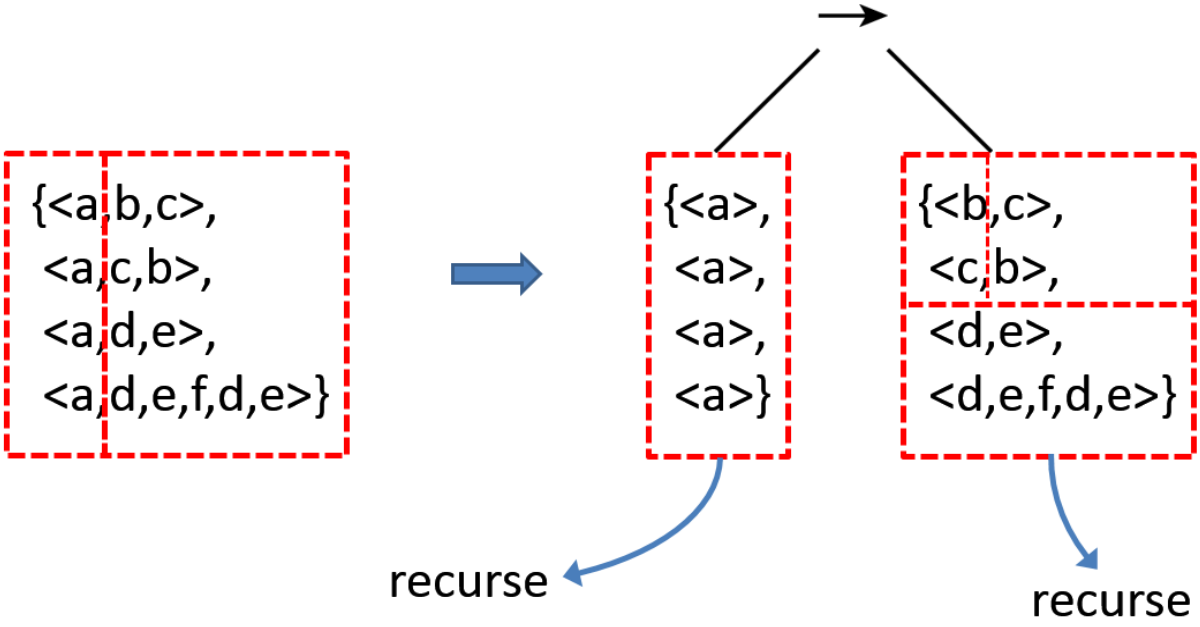

Figure 4: Inductive Miner cut points example[2]

The authors specify 1 major limitation, - the log should contain enough traces of activities execution behaviour. However, in the improved version Inductive Miner incompleteness (IMin) this limitation was lifted [5]. IMin is more complex algorithm than IM, where a simple activities partition step was replaced with an optimisation problem. The improved version estimates probabilities of the activities relations according to a predefined relationship formulas, and searches for a partition with the highest score. Authors performed a series of test, and concluded that Inductive Miner incompleteness is able to discover correct process models even from a small incomplete logs, and require less information than other process discovery algorithms.

Heuristics Miner [11] is one of the few algorithms, which can mine process models from incomplete logs and is robust with respect to noisy logs. It is a control-flow mining algorithm, which first builds event dependency graph (DG), to analyse causal dependencies of events. The relations of event pairs are ranked by frequency-based metric, indicating the certainty of dependency relation between two events, and combined into a directed connected graph.

[2] Process Discovery: Inductive Miner

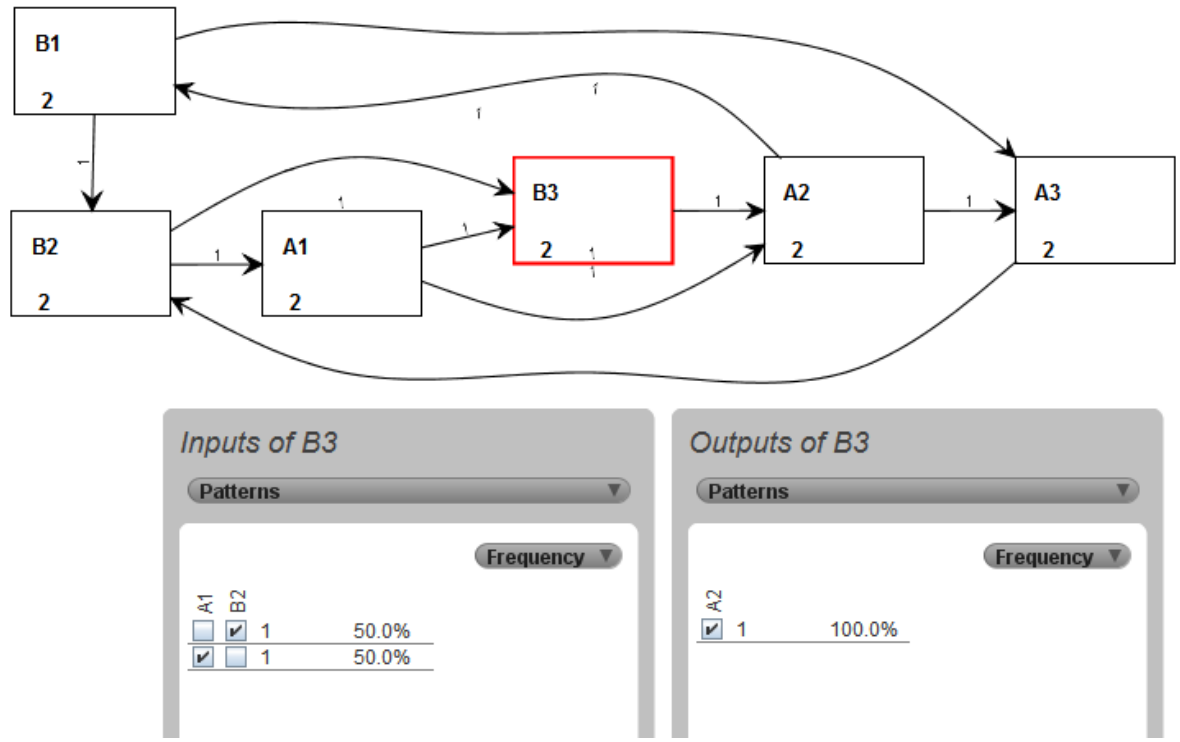

Figure 5: Heuristics Miner Dependency Graph example

Figure 5 shows a Dependency Graph example, where dependencies represented as edges and activities as nodes, and the strongest relations of event B3 inputs and outputs. Number inside each event box indicate the task frequency, while the numbers on arcs indicate the dependency relation reliability.

To obtain a process model Heuristics Miner extends DG into an internal representation called augmented Causal Net, which allows to mine splits and joins. Finally, the algorithm analyses possible extensions of the process model, i.e. long-distance relationships, graph optimisation and pruning, etc.

2.4 Related research

There exist a lot of different process mining algorithms, which could be classified as discovery of procedural (structured) and declarative (unstructured) process models, according to [12]. Nowadays, procedural mining is a dominant process discovery technique, but processes, with high level of variability, could be represented with declarative language in a more compact way. The approach proposed by the authors of [13] combines the two techniques, in order to capture strictly structured and non-structured flexible blocks, and represent them in a single process model.

To discover hybrid process models, the authors employ divide & conquer technique. First, they analyse the log in order to find events within structured and unstructured contexts, or as authors refer to it, - context analysis technique. Then, having these sets of events, the log is split into several pieces containing only either structured or unstructured events. Such division allows to employ appropriate mining algorithms, in particular, Inductive Miner for structured events and Declare Miner for unstructured. In the end, a hierarchical top-level process model is mined, usually represented as a structured process, with pseudo-activities representing child sub-processes.

To represent the final hybrid model, recently 2 extensions to an imperative model presentation formats were proposed: R/I-net and BPMN-D. Authors of the first one [14] expressed Declare semantics in the form of Petri nets, where Declare constraint is mapped to a Petri net fragment with weighted, reset and inhibitor arcs. The authors of the latter [15] introduce Declare constructs into BMPN, by extending activity nodes and sequence flow arcs, preserving backwards compatibility.

Authors of the BPMN Hierarchical Miner [16] used modularity or hierarchy, e.g. notions of parent process and groups of repetitive events, called sub-process, to represent process models with highly complex structures. The proposed approach has 3 steps. At first, sub-processes are identified. Using clustering techniques on event attributes, or analysing other dependencies of attributes combinations, unique event blocks are identified. Next, log is divided into smaller pieces according to event blocks of corresponding sub-processes. This allows to mine and discover a process model for each sub-process, with existing procedural (flat) mining algorithms, namely Inductive Miner. Lastly, knowing a hierarchy of logs, process model hierarchy is generated. Heuristics analysis of root log allows to identify boundary events, event sub-processes and markers, which then makes it possible to glue all discovered models together. There are 2 known limitations described by the authors: log should be noise-free and requires correctly assigned attributes for each event.

3 Contribution

In this chapter, we describe the main contribution of this thesis. We first introduce some definitions and theorems useful to understand the proposed discovery algorithm. Then the algorithm itself is described in detail. Finally, a discussion about advantages and limitations of the proposed algorithm concludes the chapter.

3.1 Definitions

Definition 1 (block events): set of events between corresponding split and join gateways.

Definition 2 (start / end events of a block): subset of the block events, which are directly connected to an opening / closing split gateway.

Definition 3 (unassigned events of a block): subset of the block events, which are known to belong to an AND gateway (parallel) block (see also Theorem 1). However, these events could belong to any branch of the parallel block, because a branch assignment cannot be determined (see also Theorem 2).

Definition 4 (incomplete log): log generated starting from a process model, which does not contain the complete set of all the possible paths ([2], [5]). For example, a process model consisting of 10 activities, which are executed in parallel, would have 10! (3,628,800) possible combinations.

Definition 5 (context analysis): technique that allows us to determine the set of immediate predecessors and successors for each event in the log.

Definition 6 (filtered log): log derivative, in which only a selection of events is kept in every trace. As an example, let us consider a log L, with 2 traces and 3 activities. The filtered log $L_{\{B,C\}}$ would contain the same amount of traces, preserving the order, but containing only events B and C.

Original log L

A, B, C;
B, A, C;

Filtered log $L_{\{B,C\}}$

B, C;
B, C;

Definition 7 (placeholder log): log derivative, in which some sequences of events are replaced with a pseudo-event (a placeholder). In particular, only the first occurrence of the sequence is replaced by the pseudo-event, whereas the other events are removed. As an example, let us consider a log L, with 2 traces and 3 activities. The placeholder log $L^x_{\{B,C\}}$ would have the first occurrence of events B or C substituted by a placeholder X in every trace. The other occurrences of B and C are skipped.

Original log $L_{\{A,B,C\}}$

A, B, C;
B, A, C;

Placeholder log $L^x_{\{B,C\}}$

A, X;
X, A;

Theorem 1 (grouping parallel block events): it is guaranteed that if an event belongs to a parallel block, then all its predecessors belong to the same block in case we are trying to identify the first block of a model. For identifying the following parallel blocks, we can iteratively group together events belonging to the same block based on the fact that if an event belongs to a block, then all its predecessors belong to the same block unless they are already tagged as belonging to a previously identified block.

In a log, the branches of a parallel appear mixed with each other, meaning that predecessors of one branch start events could be events of another branch. Knowing the events that belongs to previously identified blocks, this property allows us to group together events belonging to a parallel block, knowing only the start events of the block. Proof of this theorem reuses the definition of concurrently executed events [17], but instead of considering branches with a single activity, we consider multiple events per branch.

To illustrate the theorem, let us consider the process model in Figure 6.

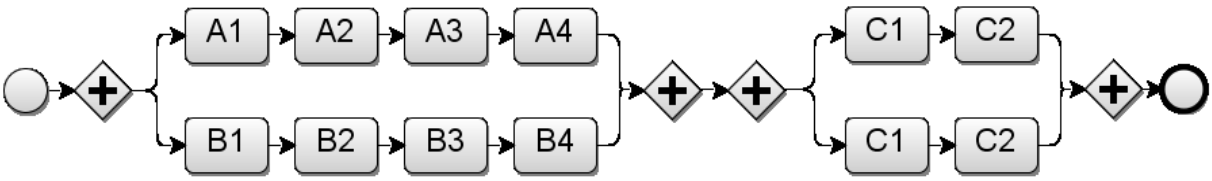

Figure 6: Theorem 1 example process model

A log of two traces would be enough to find the events belonging to each block.

```
T1: A1, A2, A3, A4, B1, B2, B3, B4, C1, C2, D1, D2;
T2: B1, B2, B3, B4, A1, A2, A3, A4, D1, D2, C1, C2;
```

Events A1 and B1 are start events of a block, and there are no events belonging to previous blocks. According to the theorem, predecessors of events belonging to the block belong to the same block. Let us focus on T1, events A1, A2, A3 and A4 are predecessors of B1, thus they belong to the same block as B1. The same idea applied to T2 (using A1 as an anchor), allows us to identify B2, B3 and B4 as members of the same parallel block. Next, we look for predecessors of the newly found events, but, in this case, there are no new block events to be identified.

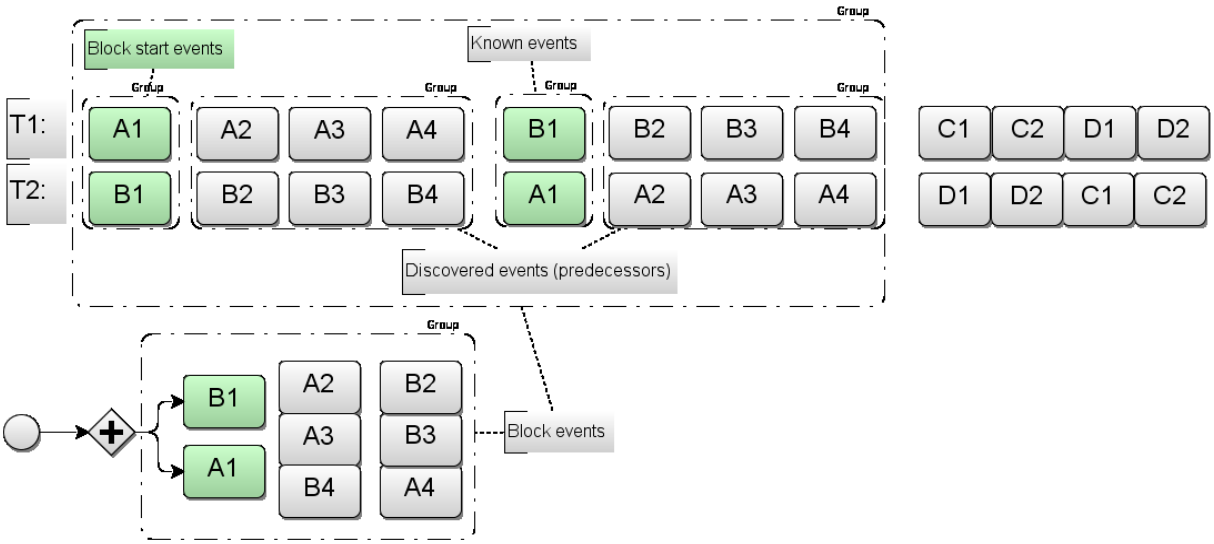

Figure 7: Theorem 1 block events discovery

After having identified events belonging to a parallel block, they are tagged and used as prior information for the identification of the next block. Events C1 and D1 are the start events of the following block. Applying the same procedure to the second block, we can identify its events.

The example traces T1 and T2 are extreme cases, which reveal the structure of every branch. The theorem proves to be also useful for the discovery of process models from logs with highly intermixed parallel branches. For example, let us consider another log of two traces:

```
T1: A1, A2, B1, A3, B2, A4, B3, B4, C1, C2, D1, D2;
T2: B1, A1, B2, A2, B3, A3, B4, A4, D1, D2, C1, C2;
```

In this example, the following steps are performed to identify events of the first block:

Table 1: Steps performed to identify block events

Identified events	**Action**	**Block events**
A1, B1	A1 and B1 are start events of the block.	A1, B1
A2	A2 is a predecessor of B1 in T1	A1, B1, A2
B2	T2: … B2, A2, …	A1, B1, A2, B2
A3	T1: … A3, B2, …	A1, B1, A2, B2, A3
B3	T2: … B3, A3, …	A1, B1, A2, B2, A3, B3
A4	T1: … A4, B3, …	A1, B1, A2, B2, A3, B3, A4
B4	T2: … B4, A4, …	A1, B1, A2, B2, A3, B3, A4, B4
{}	No new predecessors, stop iterating	A1, B1, A2, B2, A3, B3, A4, B4

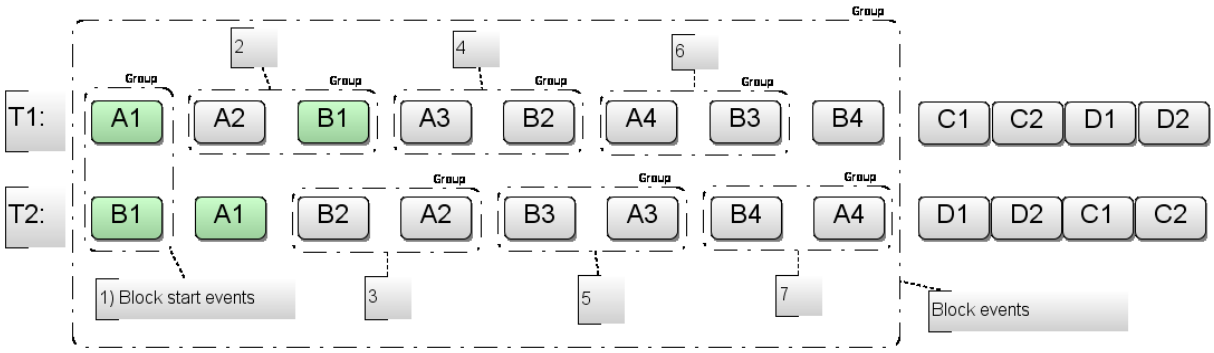

Figure 8: 2 trace predecessors chaining example

Figure 8 visualizes how knowledge about parallel block events percolates. Iteration repeats until no more events belonging to the block can be found. We assume that in a log all the interleavings needed to assign an event to the corresponding block are available. Thus, in some cases, we find less events than the block actually has, although we are never overestimating them (i.e., finding events, which do not belong to the block). For example in Figure 8, if we swap the position of B4 and A4 in T2, we would not have enough information to assign B4 to the first block. We therefore would discover the model: Seq(Block1, B4, Block2).

Theorem 2 (identifying branches in a parallel block): after having identified start events of each branch in a process model, and the block events belonging to a parallel block, it is guaranteed that predecessors of a branch start event do not belong to the same branch. When there are several start events in a branch, then the rule applies to the first observed start event in a trace, see Figure 31 for an example.

For any trace in a log, a non-start event can appear only after a branch-matching start event. It also means that the predecessors of one branch start event, are members of other branches. Proof of this theorem reuses directly follows and transitively follows theorems, initially proposed in [18], but scaled with respect to concurrency of a parallel block.

To show an example, let us consider the process model in Figure 9.

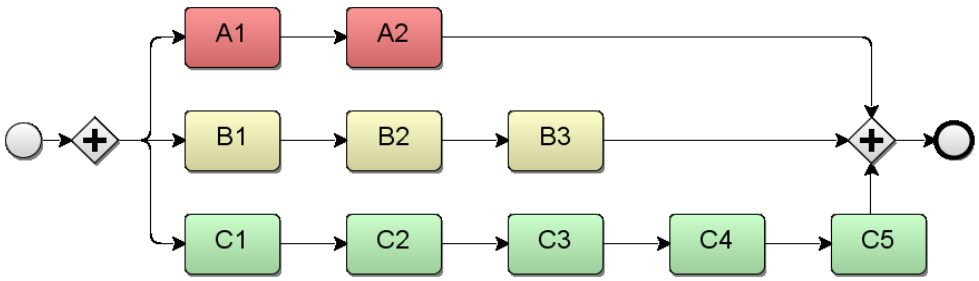

Figure 9 Open branches example model

A*, B* and C* events are grouped by colour and name prefix, and belong to the same branch. Suppose that we know that A1, B1 and C1 are start events of the parallel block, consider the following trace (which is compliant with the model):

```
T1: C1, C2, C3, B1, C4, B2, A1, A2, B3, C5
```

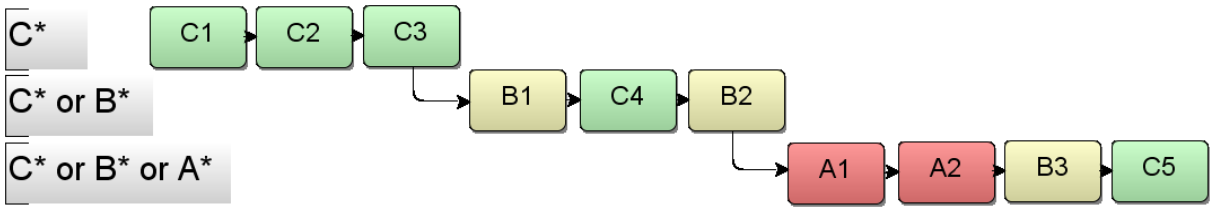

Figure 10 Example trace

Processing the trace from left to right, the following steps are performed:

Table 2: Theorem 2 steps

Event	Action	Open branches
C1	C1 is a start event, add to open branches.	{} + { C* } => { C* }
C2	Map C2 to open branches. C2 -> { C* }	{ C* }
C3	C3 -> { C* }	{ C* }
B1	B1 is a start event, update open branches.	{ C* } + { B* } => { C*, B* }
C4	C4 -> { C*, B* }	{ C*, B* }
B2	B2 -> { C*, B* }	{ C*, B* }
A1	A1 is a start event, add to open branches.	{ C*, B* } + { A* } => { C*, B*, A*}
A2	A2 -> { C*, B*, A* }	{ C*, B*, A* }
B3	B3 -> { C*, B*, A* }	{ C*, B*, A* }
C5	C5 -> { C*, B*, A* }	{ C*, B*, A* }

After trace analysis it is obvious that everything between C1 and B1 belongs to C*, but not to A* or B*, because only C1 start event had appeared. Thus from the given trace one could deduce that:

- [C2 ... C3] belong only to C* branch
- [C4 ... B2] belong either to C* or B* branches, but not to A*
- [A2 ... C5] could belong to any branch

Note that there are some events that for sure belong to a specific branch. For example C2 and C3 belong to branch C* in Figure 10. Therefore the theorem above is useful to identify only these events. This theorem does not allow us to assign the other events to a specific branch.

3.2 Algorithm description

The algorithm tries to reconstruct a process model sequentially, from left to right. The idea is to identify and process the log in blocks, i.e., group of events between gateways. Each identified branch of a block is isolated into a sub-log and mined separately. Figure 11 gives a high level overview of the algorithm.

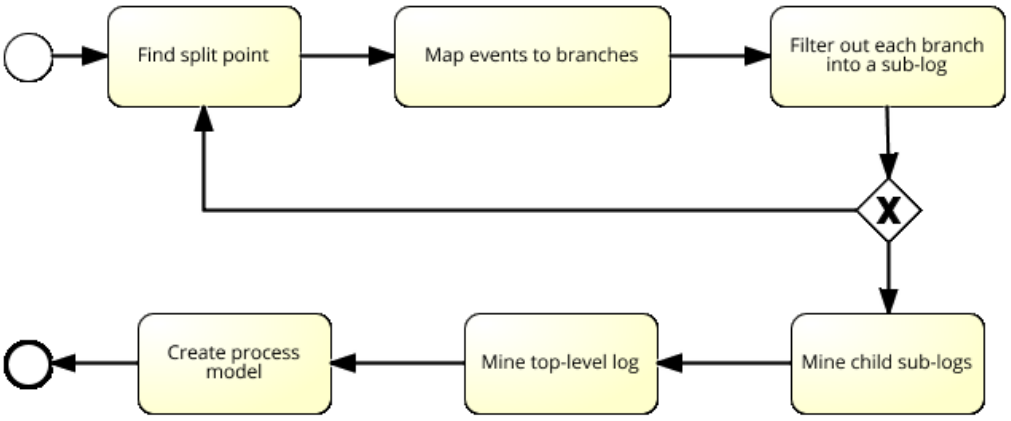

Figure 11: algorithm overview

We first try to identify initial activity of a process model, if it is always the same, then we found an activity of a sequence, and continue analysing the next one. However, when initial event differs from trace to trace, then we assume that a split point exists, and the events appearing at this position are start events of a block. A split point in a log corresponds to an opening gateway in the process model. The algorithm considers XOR and AND gateway types. To identify the gateway type we filter out start events (see definition 2), and mine it with the Inductive Miner.

XOR blocks have an exclusive execution of branches, meaning that for each trace only one branch of the block is observed. This property allows us to discover groups of events belonging to each branch, and the whole block. The Inductive Miner is able to mine XOR blocks properly, thus no further processing is needed. The discovery of complex AND blocks is more challenging with the Inductive Miner.

Branches of AND blocks are executed concurrently, meaning that in each trace all branches of the block are observed and the different branches could appear mixed with each other. We use Theorem 1 to identify the group of events belonging to the same block. Next, we identify events belonging to each parallel branch. To this aim, first, we apply Theorem 2 , then we use Heuristics Miner to map the rest of unassigned events (remember that Theorem 2 is not always able to completely identify all the branches).

After having identified the block events and branch mapping, we create a placeholder log in which events belonging to each branch of each parallel block are replaced by a placeholder (see definition 7). This is done iteratively until all the blocks in the model have been identified from left to right (see iteration in Figure 9). For the final part of creating the process model, we recursively apply the algorithm to each filtered sub-log representing a separate branch, and mine it with the Inductive Miner. Then, all the branches are being recursively merged into a parent process model (bottom up approach) up to the top-level.

3.2.1 Step by step walkthrough

To describe each step of the algorithm in detail, suppose that we have the following log (example S1):

```
T1: S1, A1, B1, A2, B2, A3, B3, S2, C, End;
T2: S1, B1, B2, A1, A2, B3, A3, S2, D, End;
T3: S1, A1, B1, B2, B3, A2, A3, S2, C, End;
```

Step 1 (find split point): Our algorithm is based on a technique called context analysis. In particular, it iterates over all traces of the log, and constructs a set of immediate predecessors

and successors for each event. Iterating the immediate successors from the process start, allows us to discover fixed-position events, as well as points of variability. Meaning, that event with more than 1 successor is considered a split.

For our example S1, context analysis would construct the following successors table:

```
__process_start__: { S1 }, S1: { A1, B1 }, … End: { __process_end__ }
```

We sequentially iterate from process start to process end, or until a split point (more than 1 successor) is found.

```
__process_start__ -> S1 -> { A1, B1 }
```

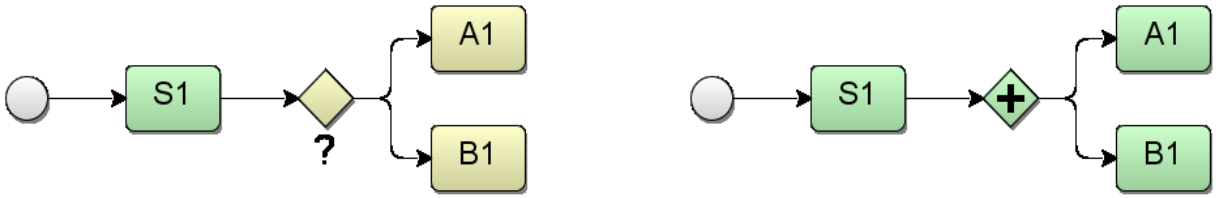

Figure 12: Context analysis finds and analyses gateway splits

In our example, S1 is the first event in all the traces and, after event S1, we have a split, where A1 and B1 are start events of a block. To identify the block type we isolate start events A1 and B1 into a filtered log (see definition 6), and mine the log with the Inductive Miner. As a result we obtain not only the block type, but also the structure of the block branches. See Figure 31 for an example of complex structure with embedded blocks mined from start events.

The result is a parallel block, and the top-level process model is:

```
Seq( S1, And( A1, B1 ) )
```

We need now to know which events belong to the newly discovered block. We use Theorem 1 to identify the group of events that belong to the block. According to the theorem, predecessors of known parallel block events A1 and B1, also belong to the block, except for events preceding the split. There is only 1 event preceding the split – S1.

The following steps are performed:

Table 3: Block events identification steps

Identified events	**Action**	**Block events**
A1, B1	A1 and B1 are start events of the block.	A1, B1
B2	B2 is a predecessor of A1 from T2	A1, B1, B2
A2	T1: … A2, B2, …	A1, B1, B2, A2
B3	T3: … B3, A2, …	A1, B1, B2, A2, B3
A3	T1: … A3, B3, …	A1, B1, B2, A2, B3, A3
{}	No new predecessors found	A1, B1, B2, A2, B3, A3

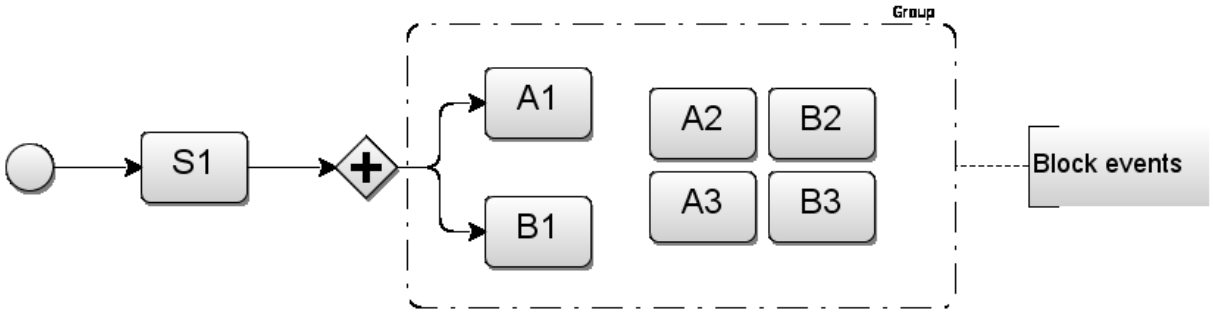

Figure 13: Block's events detection, knowing start events

Predecessor detection technique works well with branch overlapping, often observed in logs of highly structured processes, because it doesn't try to understand the internal structure of the block.

Step 2 (map events to branches): there are 2 techniques used to assign each event of the parallel block to a branch. First, Theorem 2 is used, then remaining unassigned events are mapped via Heuristics Miner.

From the previous step we know that there are 2 branches, which start with events A1 and B1. According to Theorem 2, all the events appearing in the log between A1 and B1 could be exclusively mapped.

```
T1: … A1, B1 …
T2: … B1, B2, A1 …
T3: … A1, B1 …
```

After analysing the log, event B2 is mapped to the branch starting with B1. In the 2nd trace, B2 appears after B1 but before A1, meaning that it could not belong to the same branch as A1. The rest of unassigned block events B3, A2 and A3 have to be mapped with Heuristics Miner.

Whereas Theorem 2 relies on edge cases to reveal branch content, the Heuristics Miner builds a dependency graph of observed paths, and relies on the most common patterns.

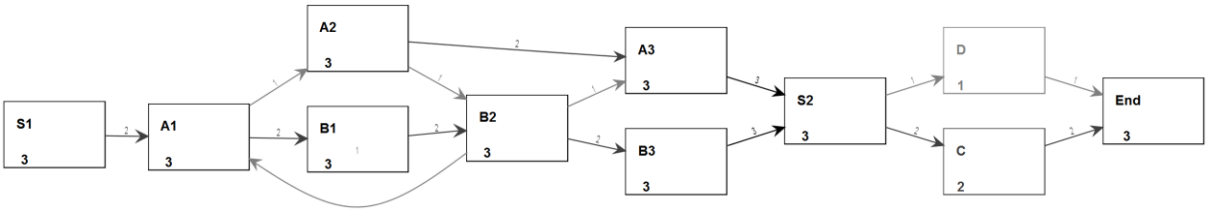

Figure 14: Heuristics Net (Dependency Graph)

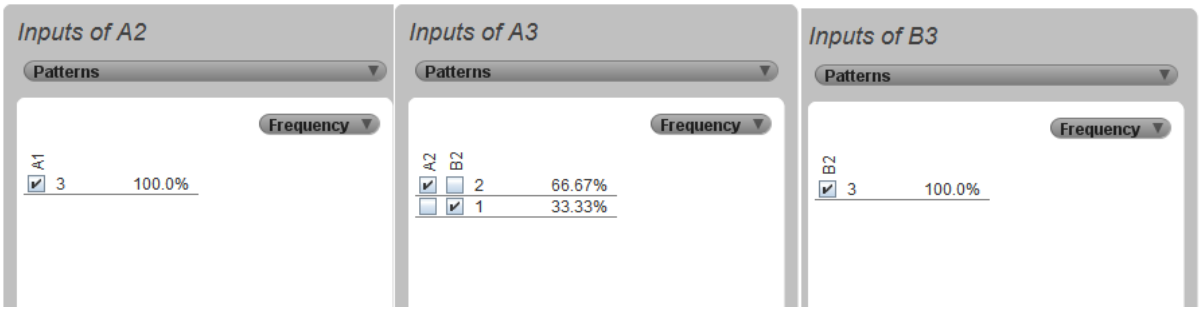

Figure 15: Dependency cause tables for events A2, A2 and A3

Figure 15 shows the input patterns of the unassigned events A2, A3 and B3, produced by the Heuristics Net. We assign each event to a predecessor with the highest frequency.

Table 4: Unassigned events mapping steps

Unassigned event	Action	Branch mapping
A2	A1 is the sole input of A2, thus belong to the same branch	A2 → A1
A3	A2 has the highest frequency	A3 → A2 (A3 → A1)
B3	B2 with 100%	B3 → B2

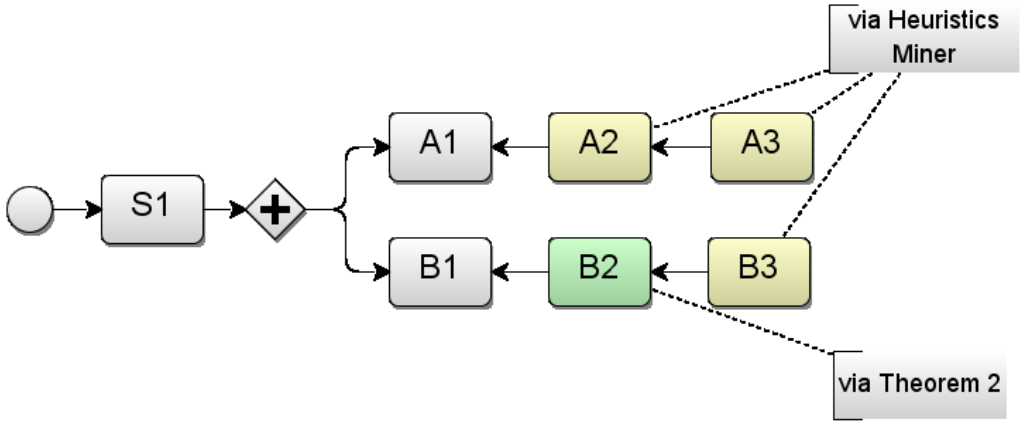

Figure 16: Unassigned events mapping to branches

As a result, we have mapped each event of the block to a branch, but the internal structure of each branch remains unknown.

Step 3 (filter out each branch in a sub-log): From the previous step, we have group of events belonging to each branch. We now create a placeholder log (see definition 7) where we use a placeholder log instead of the original one and for each branch we create a filtered sub-log (see definition 6).

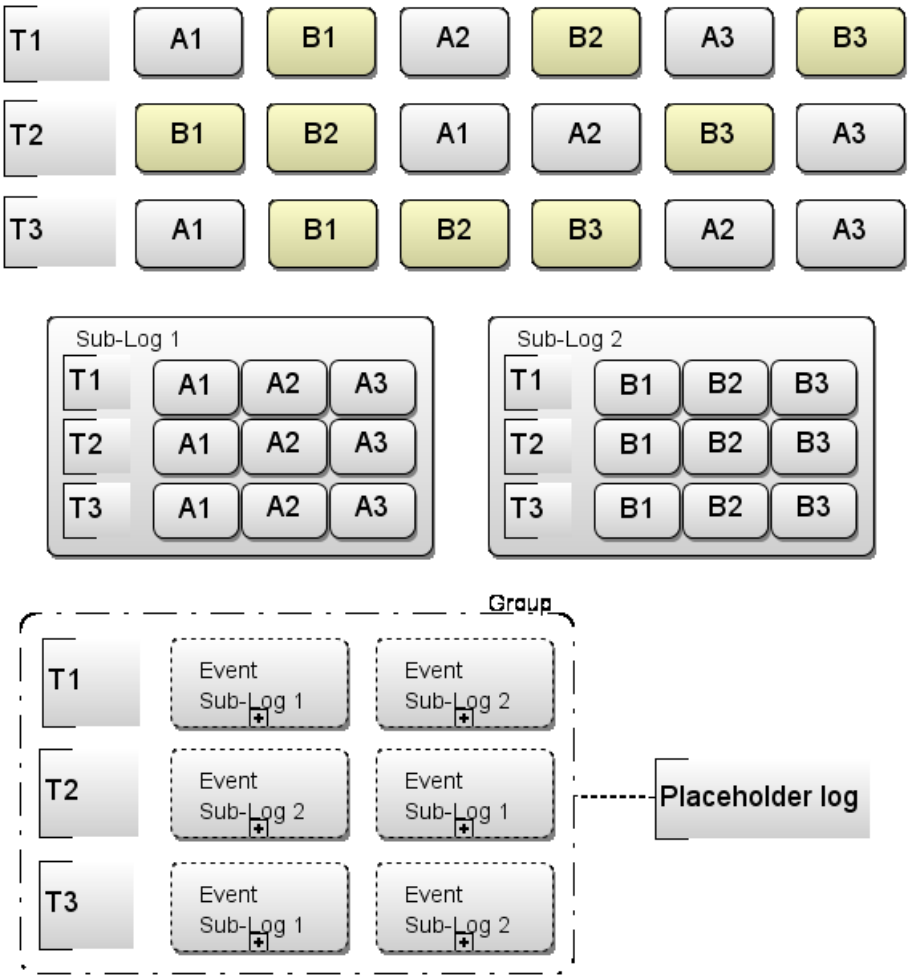

Figure 17: Example of branch separation into sub-logs

Each branch sub-log and placeholder log should contain the same amount of traces as the top-level log, and preserve the relative order of events. Filtered logs Branch_A and Branch_B are:

$$L_{\{A1,\ A2,\ A3\}}$$

```
T1: A1, A2, A3;
T2: A1, A2, A3;
T3: A1, A2, A3;
```

$$L_{\{B1,\ B2,\ B3\}}$$

```
T1: B1, B2, B3;
T2: B1, B2, B3;
T3: B1, B2, B3;
```

The placeholder log, with pseudo-events Branch_A and Branch_B (substituting by first occurrence), is (placeholder log S1):

```
T1: S1, Branch_A, Branch_B, S2, C, End;
T2: S1, Branch_B, Branch_A, S2, D, End;
T3: S1, Branch_A, Branch_B, S2, C, End;
```

The filtered out logs should be linked to an appropriate pseudo-events, and kept for further processing. At this point, the current block is considered identified, and we can continue processing the top-level log with context analysis.

Step 4: analyse events following the newly discovered block (repeating steps 1-3). From step 1, with we found the following top-level structure:

```
Seq( S1, Block_1 )
```

We continue looking for the next split with context analysis.

```
__process_start__ -> S1 -> { Block_1 events } -> S2 -> { C, D }
```

First, we find a fixed-position activity S2, then a new block with start events C and D. To identify block type, the algorithm filters out start events C and D.

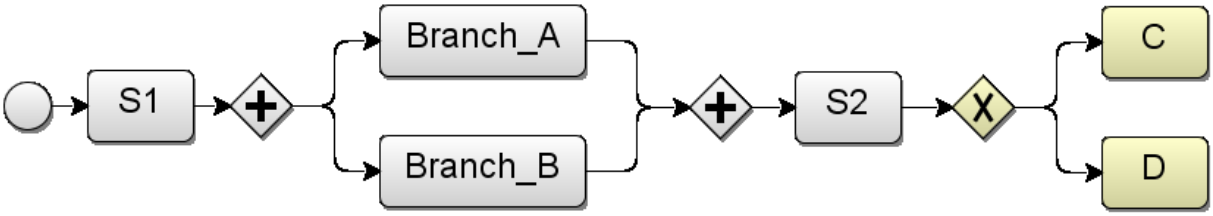

Figure 18: Illustration of sequential of model reconstruction

The result is a XOR gateway, thus we use the Inductive Miner to find other events, which belong to the block. In this case, the block has only 2 events.

Finally, the context analysis discovers the last activity of the top-level process:

```
Seq( S1, And( Branch_A, Branch_B ), S2, Xor( C, D ), End )
```

Step 5 (create a process model): we discover the underlying structure of each extracted branch, and if necessary, we recursively apply the algorithm. Then, we merge the child process trees into a top-level tree.

Logs $L_{\{A1,\ A2,\ A3\}}$ and $L_{\{B1,\ B2,\ B3\}}$ from step 3 are mined with the Inductive Miner, and the following results are obtained: Seq(A1, A2, A3) and Seq(B1, B2, B3).

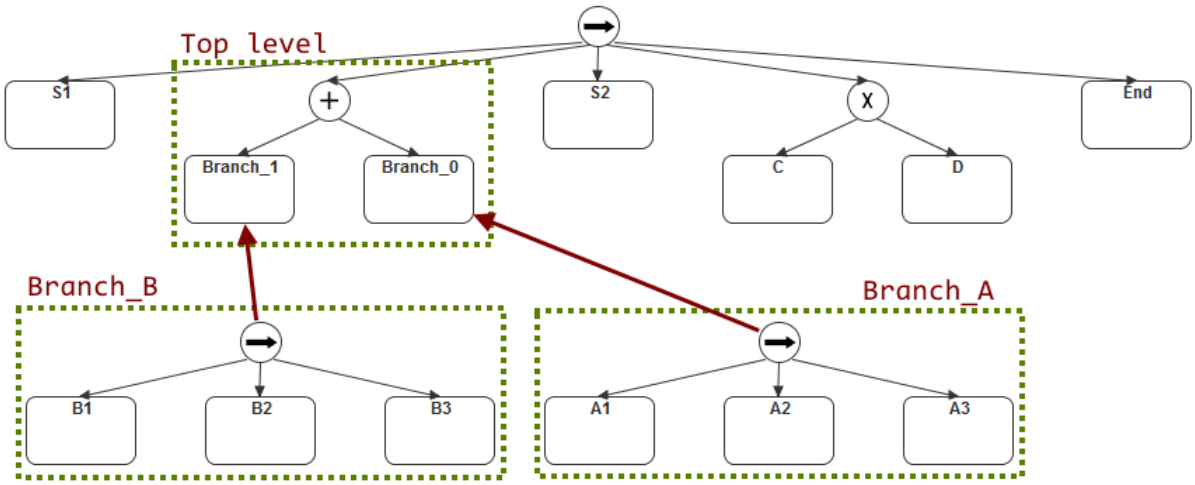

Figure 19: Final process tree composition example

Figure 19 illustrates the replacement of the placeholder activities of the top-level tree with the child process trees. Tree composition process uses a bottom-up approach, meaning that for hierarchy with several layers the lowest level children are replaced first.

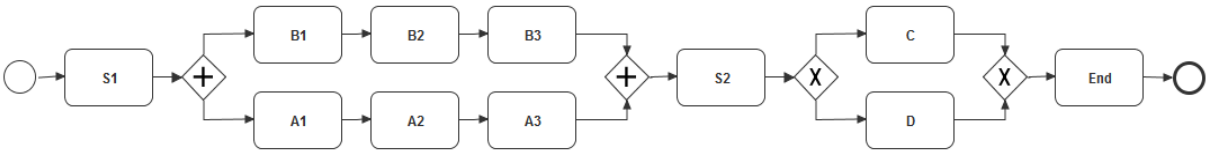

Figure 20: Discovered process model

The final process model discovered from example S1 is show on Figure 20. There are several outputs produced by the algorithm:

1. A process tree, which could be converted by existing libraries into other model representation formats.
2. Placeholder top-level log.
3. All filtered child logs, which represent extracted the branches.

3.2.2 Complex process model walkthrough

Let us review a more complex process model, without fixed-position events between parallel blocks. Suppose that we have the following log (example S2):

```
T1: A1, A2, B1, B2, B3, A4,  D1, D2, C1, E1, E2, E3, C2, C3,  End;
T2: A1, A3, B1, B3, B2, A4,  D1, D3, E1, E2, C1, C2, C3, E3,  End;
T3: B1, B2, A1, A2, A4, B3,  E1, E2, C1, E3, D1, C2, D3, C3,  End;
T4: B1, A1, B2, B3, A2, A4,  C1, C2, E1, D1, C3, E3, E2, D2,  End;
T5: A1, B1, A3, B3, A4, B2,  C1, D1, C2, E1, E3, E2, D3, C3,  End;
```

The target top-level process tree should look like (source process model is shown on Figure 27):

```
Seq( And( A*, B* ), And( C*, D*, E* ), End )
```

Step 1: using context analysis we find a split gateway and block start events A1 and B1.

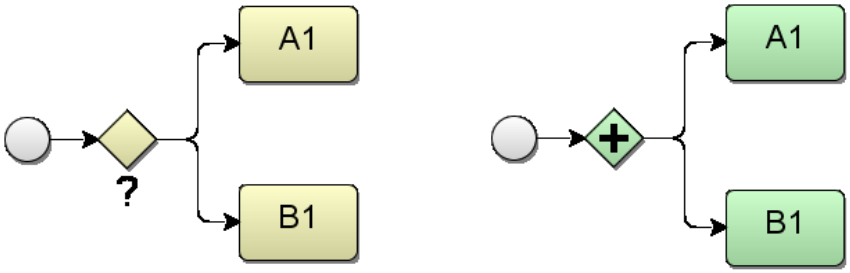

Figure 21: Context analysis of first block

Next, we mine filtered log containing start events A1 and B1 with the Inductive Miner. The discovered process tree provides information about the block type that is a parallel block.

Step 2: applying Theorem 1 we discover events that belong to the block.

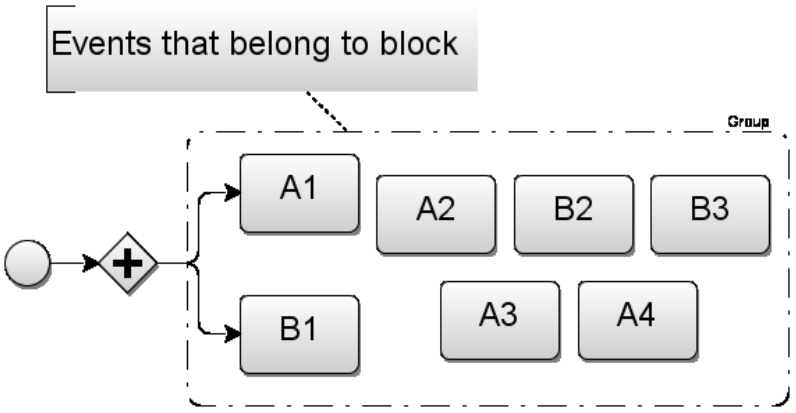

Figure 22: Set of events belonging to the block

Step 3: using Theorem 2 allows us to distinctly map events A2, A3 and B2 to branches (from T1, T2 and T3 respectively). Next, we use the Heuristics Miner to find branch mapping for the rest of unassigned events.

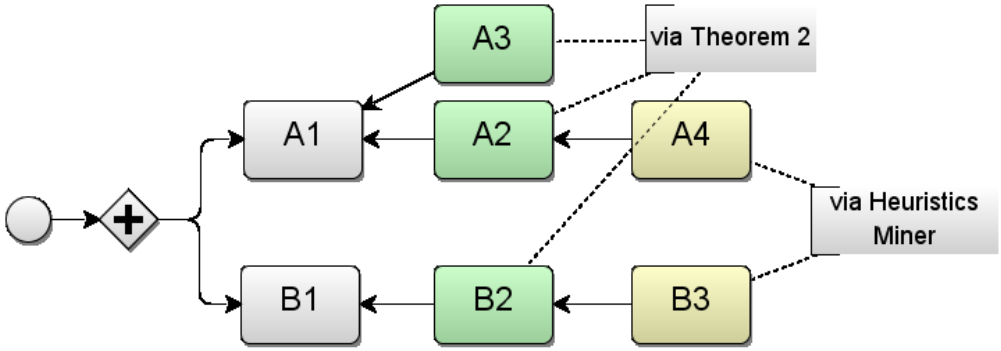

Figure 23: Event to branch mapping

Steps 4-5: we create 2 filtered out logs for each branch, and create a placeholder top-level log with pseudo-events. Then we proceed with context analysis, and discover next block's start events D1, C1 and E1. With the Inductive Miner we discover that there are 3 branches.

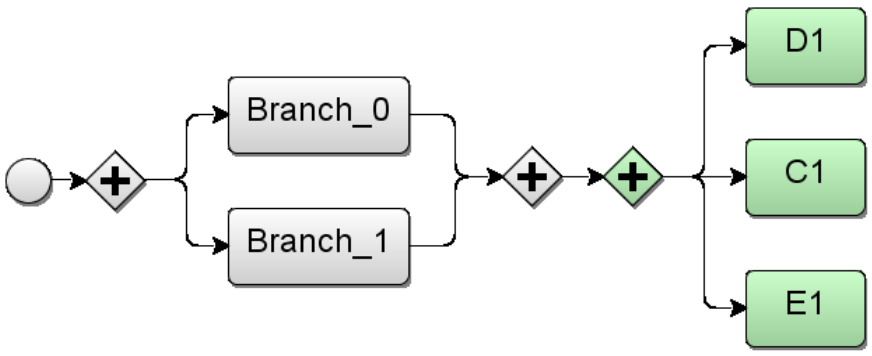

Figure 24: Context analysis with prior information

Steps 6-7: we find events, which belong to the block, then map them to the branches. From traces T1, T2, T3 and T4 we map events D2, D3, E2 and C2 respectively.

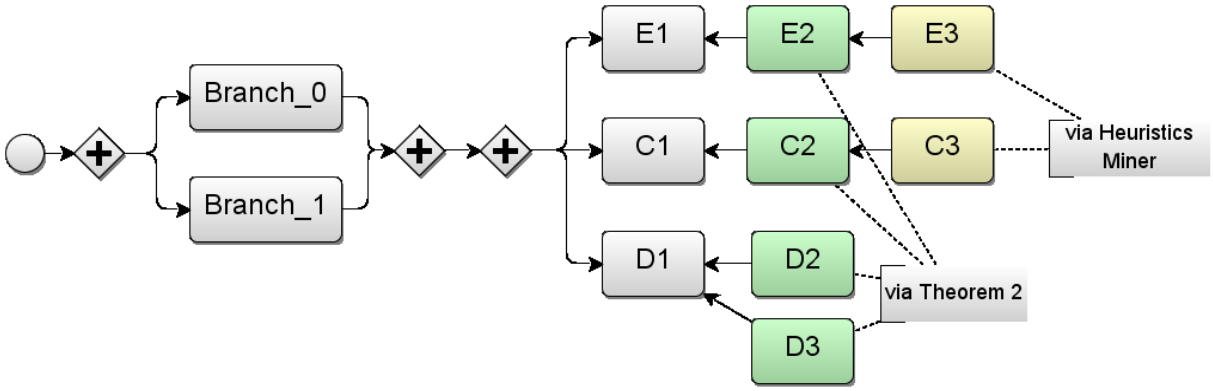

Figure 25: Event to branch mapping

Steps 8-9: we filter out branches of the 2nd block, update top-level placeholder log and perform context analysis on the remaining events.

Step 10-12: we mine a process tree from the top-level log with the Inductive Miner (the result is shown on Figure 26). Then recursively analyse each filtered log with the algorithm, and mine a respective process tree for each branch.

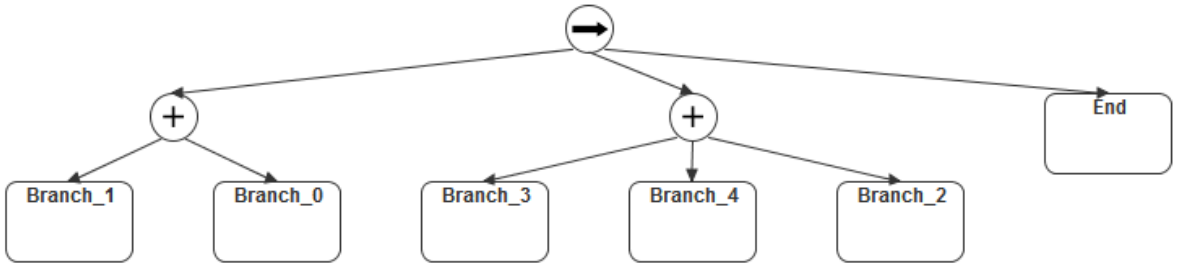

Figure 26: Top-level process tree with placeholder events

Finally, we replace the placeholder events in the top-level process tree with the discovered child process trees. The composed process tree is shown on Figure 28.

In conclusion, we compare the discovered process model (Figure 29) with the source model (Figure 27), and find no difference – they are identical.

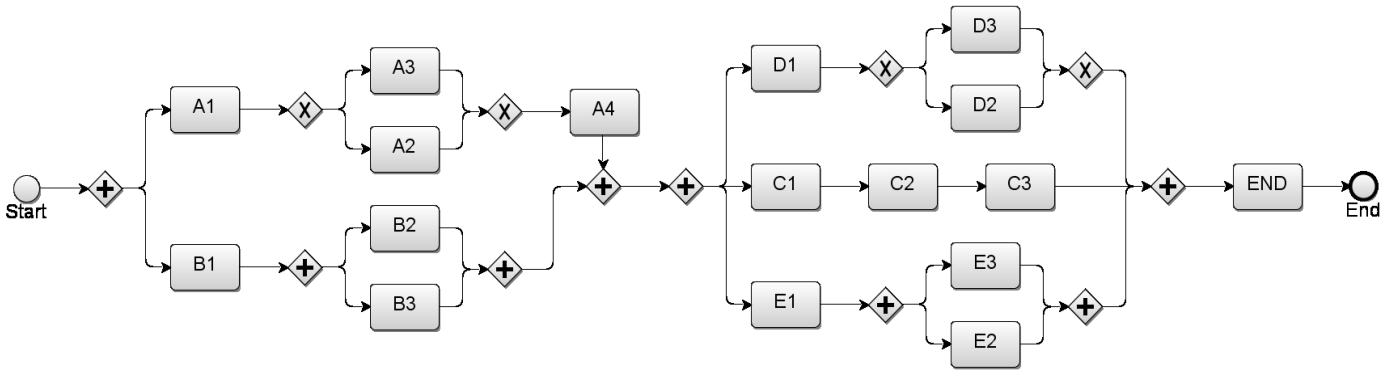

Figure 27: Source process model of example log S2

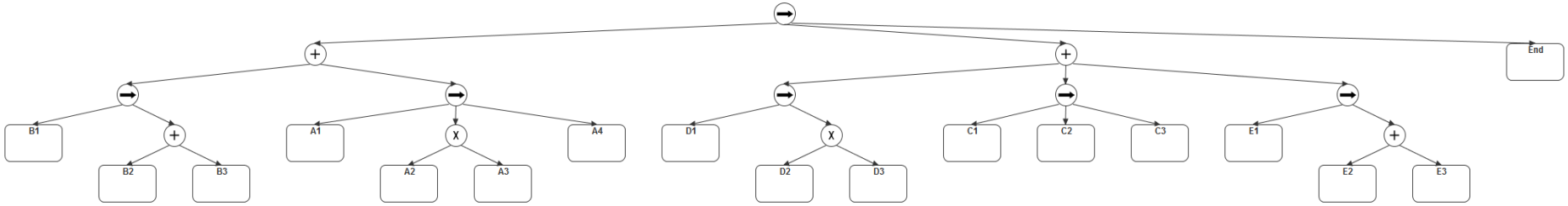

Figure 28: Discovered process tree from example log S2

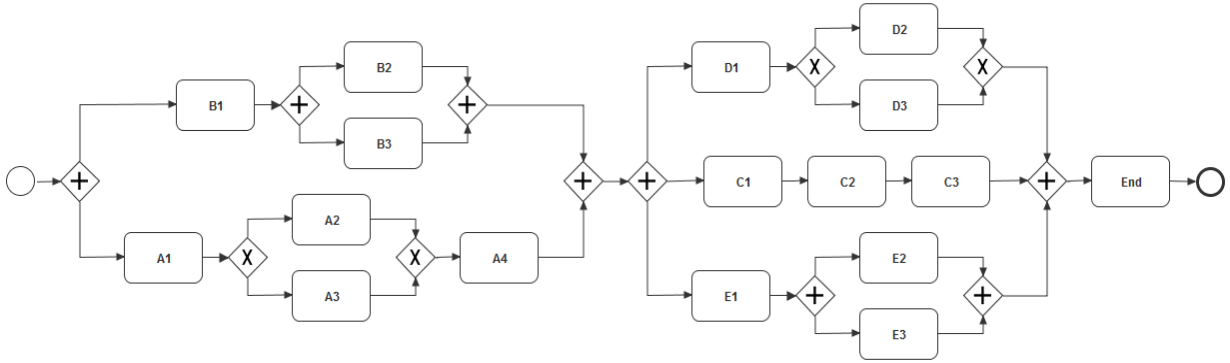

Figure 29: Discovered process model from example log S2

3.2.3 Why does the algorithm work?

Procedural models go from start to end, meaning that the process has a start and an end. Top-level process' order of execution is always constant, - sequentially, from left to right. Every trace in a log is a reflection of this principle, it is an ordered list of events which follow the process from the beginning to the end. What if all events, of each block-structure, were known? Then mining a structural process model from a log would be possible with existing algorithms, by extracting events belonging to the same block-structure, into a separate logs preserving the order of events in a trace, and mining them separately.

The idea of knowing block events allows us to mark block edges in the log, since top level block's execution order is constant – it's a sequence. For example, let all events starting with "A" and "B" would belong to 2 different blocks. Given a log of traces:

```
Start, A1, A2, A3, B1, B2, End
Start, A4, A5, B3, B1, B4, End

=> Seq( Start, Block( A* ), Block( B* ), End )
```

The clear distinction of block edges allows us to determine top-level structure, without knowing the structure of blocks. This leads to another observation, - knowing only block edges would suffice to determine the top-level structure. The example above could be represented as:

```
Start, A1, …, A3, B1, …, B2, End
Start, A4, …, A5, B3, …, B4, End

=> Seq( Start, Block( A* ), Block( B* ), End )
```

This allows us to change the requirement of knowing all events in each block to knowing only start/end events of each block, in order to find the structure of the top-level model. This could be improved even further, since end events of one block are directly followed (always) by start event of the next block. This property is symmetrical, meaning that start events of a block are always preceded by end events of previous block (or empty set in the beginning of the process). Since they duplicate each other, the requirement could be lifted to knowing only 1 set of events.

```
Start, A1, …, A3, B1, …, B2, End
Start, A4, …, A5, B3, …, B4, End

=> Seq( Start, Block( A* ), Block( B* ), End )
```

Summing up, the algorithm focuses on block order of execution, gateway splits and their branches. It first tries to understand the structure of process, from meta-information and context analysis, and then guide the mining procedure. Combination with CSP-like process mining algorithms makes it possible to discover original process models without extreme edge cases, often absent in incomplete logs.

3.3 Algorithm limitations

The following limitations are inherent to the algorithm:

- Not robust with respect to noisy logs.
- Cannot handle loops, nor duplicate events names appearing in a trace.
- Internally uses the Inductive Miner Incompleteness and inherits its limitations.
- Unassigned events could be assigned to an incorrect branch.

First and most important limitation is noise, meaning that the algorithm is precise and does not tolerate errors in a log. Having a single trace, which doesn't correspond to a source model, might result in an output, different than a source model. This limitation usually inherent to all

process mining algorithms, which deal with incompleteness. Alternative solution would be to keep only the most common paths by some threshold, but it might remove edge cases which are crucial because log is incomplete.

Next limitation concerns loops and duplicate events. Events classifier takes into account event name (and transition if needed), which means that 2 or more event occurrences, with the same name, will have a merged list of predecessors and a merged list of successors during context analysis.

The reason for loop constraint is Theorem 1 (about finding block events). The theorem states that every predecessor of parallel block events is also parallel, however loop body would be seen as a predecessor without actually belonging to the parallel block. This could be seen as a disproof of theorem, but during algorithm design phase loops became a major burden, which heavily complicated the implementation. Thus was decided to drop the support of loops in favour of simpler and guaranteed working algorithm. For example let us consider the process model in Figure 30.

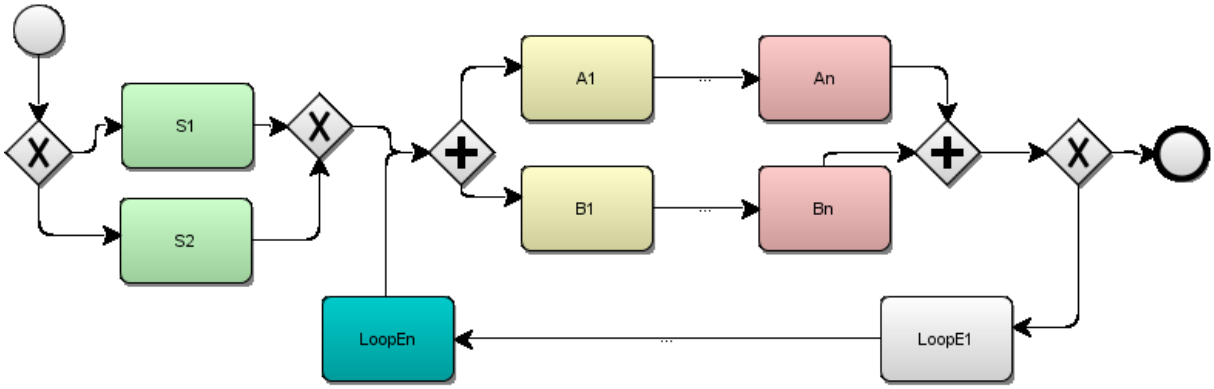

Figure 30: Loop as predecessor limitation

In Figure 30 light green events (S1, S2) are already processed events, yellow (A1, B1) and orange (An, Bn) – being processed and yet to be processed events respectively. A log corresponding to this model might contain traces like {… A1, B1 … LoopE1, LoopEn, A1, B1 …}. While processing sucha log, event LoopEn is seen as a predecessor to the parallel block, and since LoopEn does not belong to the XOR block, the theorem it will incorrectly assign it to the parallel block.

To make the algorithm work in the most possible cases, we use the Inductive Miner to determine the structure of block start events. Let us consider the example in Figure 31.

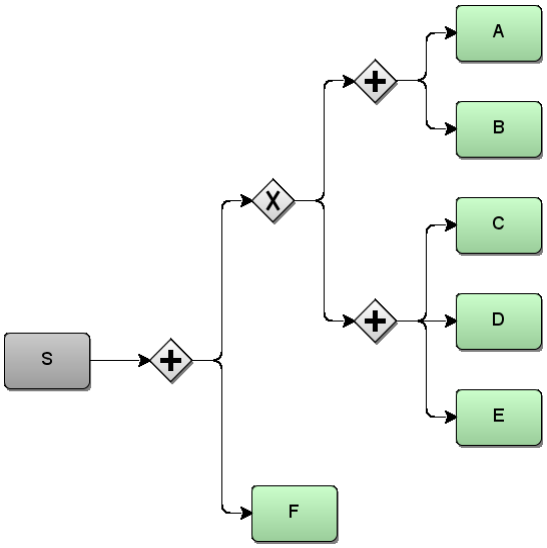

Figure 31: Example of complex structure of block start events

In Figure 31 a top-level parallel block has nested blocks, which influence the number of start events. Our goal is to determine number of block branches and group of events belonging to each of them. However this is not possible without first mining the underlying structure of each branch. Thus, the algorithm depends on and inherits limitations of the Inductive Miner.

The algorithm reconstructs the process in a sequential manner, from the beginning to the end. Thus, mistakes made during block structure reconstruction could recursively propagate to child blocks or affect the parent block.

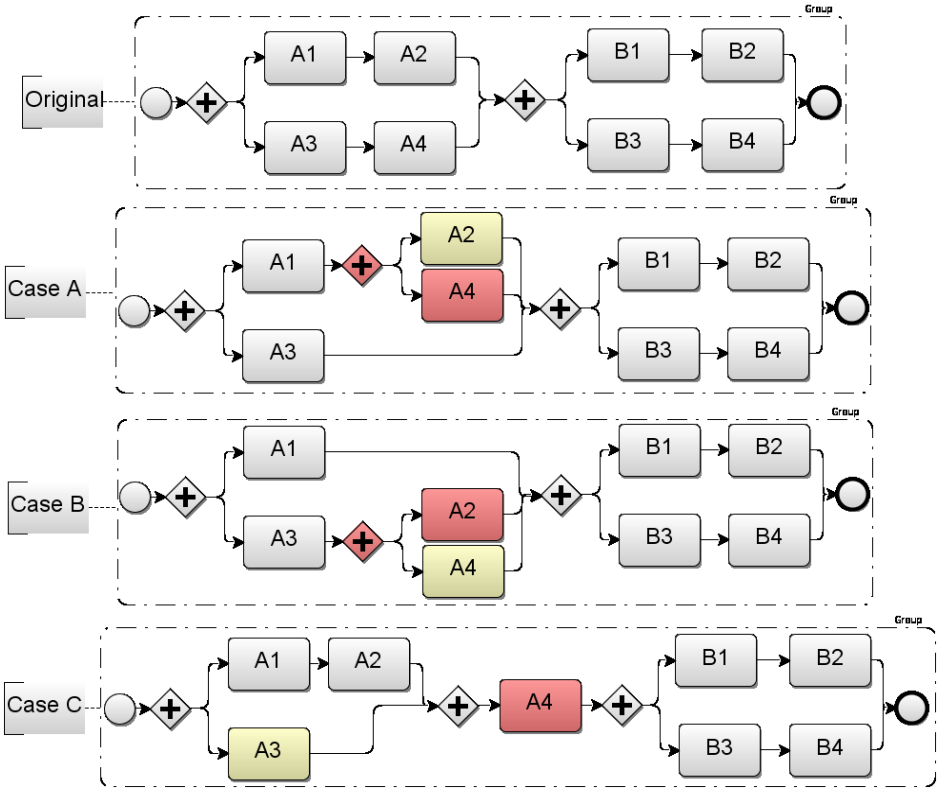

Figure 32: Examples of error propagation

Figure 32 shows original process model and 3 cases of incorrect models, which could be obtained when mining with our algorithm. Cases A and B illustrate incorrect assignment of

unassigned events to branches, and how it affects the nested blocks of the branches. The errors in these cases are caused by Heuristics Miner, i.e. the most common paths. Case C shows result of incorrect block events detection, which affects the parent block.

In general the algorithm is robust to incompleteness, but in some cases could lead to incorrect solutions. However these solutions are still suboptimal.

4 Evaluation

In this chapter, first, we compare process models produced by different process discovery algorithms from example logs S1 and S2, used in algorithm description chapters. Then we evaluate discovery of 6 artificial process models using randomly generated incomplete logs. Lastly, we measure performance of the mining algorithm.

4.1 Comparison of process discovery algorithms

The algorithm focuses on the discovery of parallel blocks from incomplete logs. In a parallel block branches are executed concurrently, they intermix creating "branch noise". This does not allow most of the process discovery algorithms to mine the correct process model without having all the edge cases.

We chose Alpha-Algorithm Miner and Inductive Miner for comparison with our algorithm, implemented as Parallel Block Miner (PB-Miner) test plugin for ProM tool[3], because the former is stable and mature, and the latter is new and flexible. Note that there are several implementation variations of Inductive Miner, but we consider only baseline and incompleteness variants.

4.1.1 Process model S1

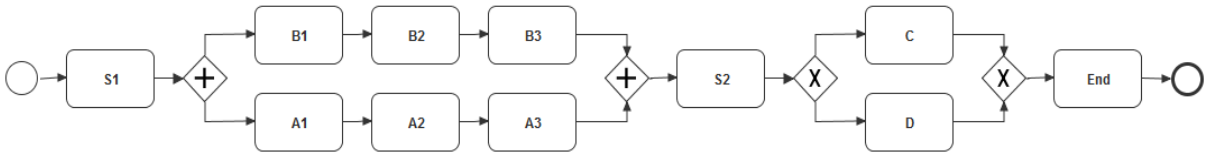

Figure 33: Process model S1

Figure 33 shows a model discovered with PB-Miner from example S1, which also corresponds to the original process model. The log consists of 3 distinct traces.

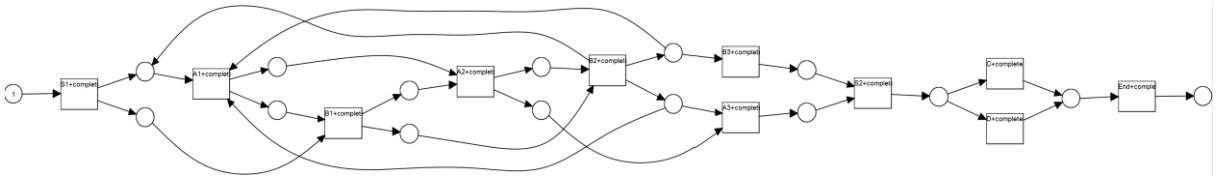

Figure 34: Model S1 discovered with Alpha-Algorithm Miner

The model discovered from the log with Alpha-Algorithm Miner is shown in Figure 34. We can see that the parallel block is not reconstructed correctly.

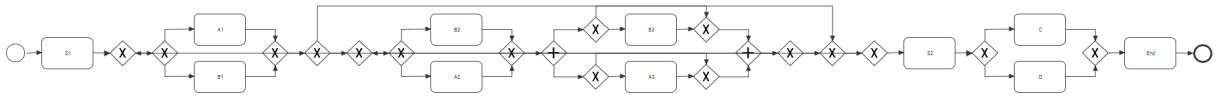

Figure 35: Model S1 discovered with Inductive Miner baseline

Figure 35 shows a model mined with Inductive Miner baseline implementation. This model is incorrect, since the parallel block is split into several distinct parts. In addition, we can see a lot of loop and event skip structures.

[3] https://github.com/sjbog/PBMiner

Alpha-Algorithm and baseline implementation of Inductive Miner did not discover a correct process model because they were designed to work with complete logs, which reveal most edge cases.

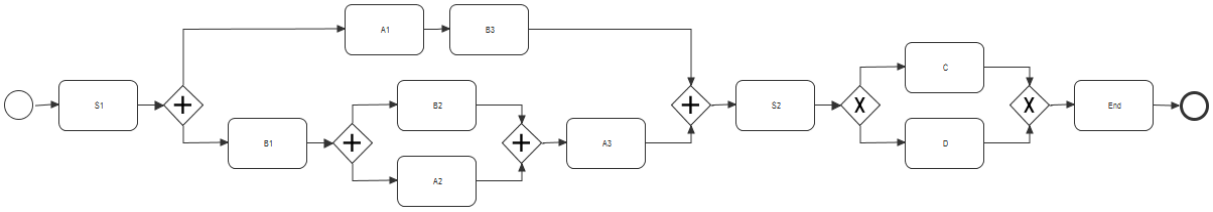

Figure 36: Model S1 discovered with Inductive Miner incompleteness

The Inductive Miner incompleteness variant, shown in Figure 36, produces an incorrect result, but close to the original. The top-level structure is perfect, however branches of the parallel block have wrong body. The miner cannot discover a correct process model because there is not enough edge cases present in the log and the log completeness is too low.

4.1.2 Process model S2

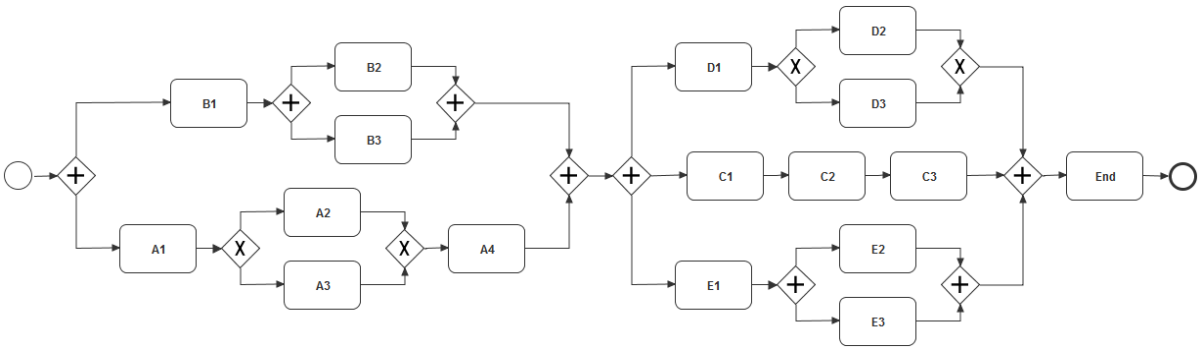

Figure 37: Process model S2

Figure 37 shows a model discovered with PB-Miner from example S2, which also corresponds to the original process model. The log consists of 5 distinct traces.

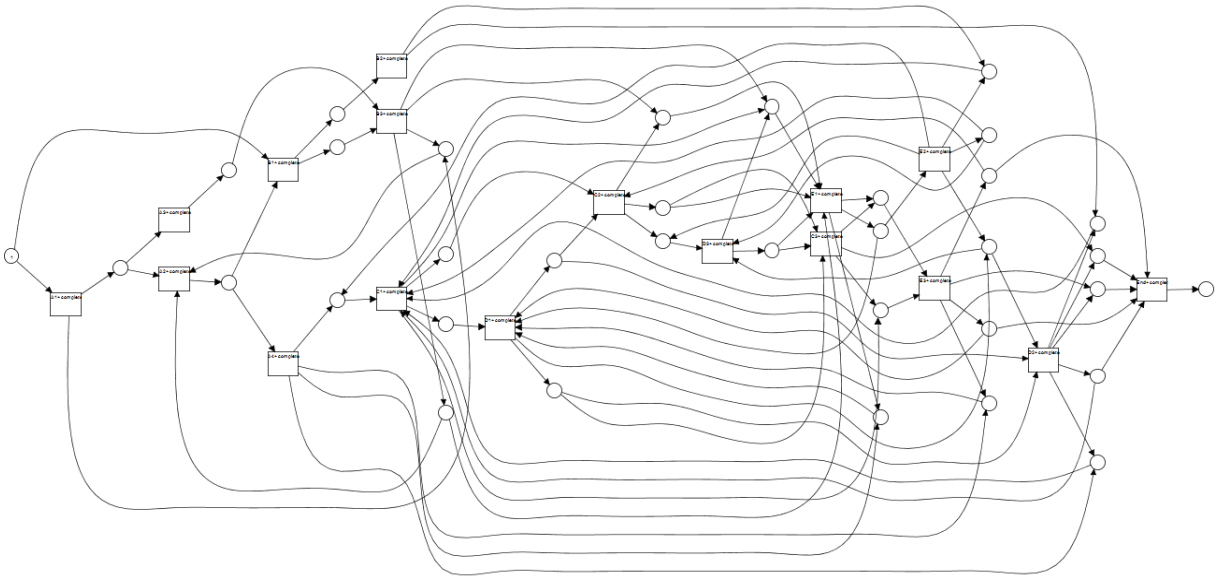

Figure 38: Model S2 discovered with Alpha-Algorithm Miner

The model discovered from the log with Alpha-Algorithm Miner is shown in Figure 38. We can see that the mined model is a "spaghetti-like" model with a lot of arcs, and does not correspond to the original process model. This result is expected, taking into account that the log has low level of completeness.

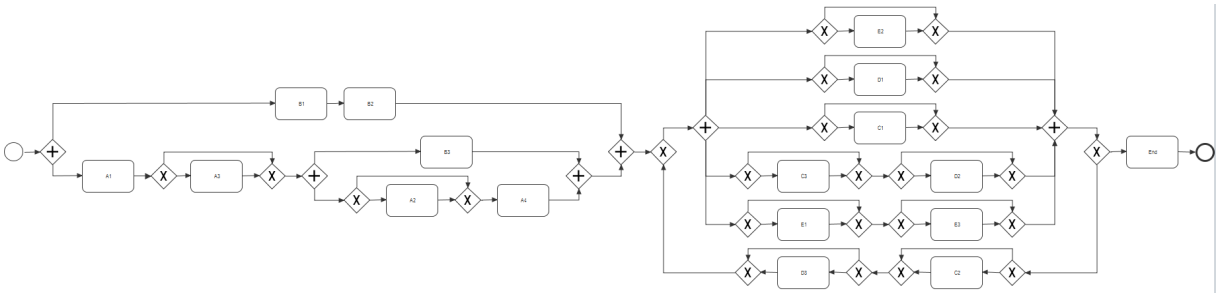

Figure 39: Model S2 discovered with Inductive Miner incompleteness

Figure 39 shows process model discovered with the Inductive Miner incompleteness variant. The result is incorrect, although the miner was able to properly discover and group together events belonging to the 2 blocks of the top-level sequence. We conclude that the log does not contain enough edge case for this level of log completeness.

4.2 Effectiveness analysis

We have used 6 process models to evaluate process discovery results. S1 and S2 are models from example logs, S3 is an artificial model, and the rest are randomly generated models. All selected models have parallel blocks with more than 5 block activities, and some have nested blocks with maximum depth of 3.

We evaluated log completeness by dividing the number of distinct traces in the log by the number of possible traces that can be generated by the model. There following formulas were used for each block:

- Sum of branch choices for XOR block (because any of the branches could be executed).
- Max of child elements for SEQ block.

- Formula $\frac{Size(\ block\ )!}{Size(\ branch1\ )! * Size(\ branch2\ )! \ldots}$ for AND block (this formula could vary depending on the nested blocks in each branch).

Let us calculate the number of possible paths for the following process models:

- Seq(A, B, C) = 1, because sequence block has only 1 way to be executed.
- Xor(A, B) = 2. Here {A} or {B} are the only choices. Each branch of the block consists of a single event, and in case of several sequential events the result doesn't change.
- Xor(Seq(A, B), Seq(C, D)) = 2 ({ A, B } or { C, D }). Each branch is a sequence with a single order of execution, meaning Sum(Seq, Seq) = Sum(1, 1) = 2.
- Xor(Xor(A, B), C) = 3, could be calculated as Sum(Sum(1, 1), 1).
- Seq(A, Xor(B, C)) = 2, with choices {A, B} or {A, C}. Calculated as Max(1, Sum(1, 1)).
- And(A, B, C) = 6, it is permutation of events A, B and C (e.g. 3!).
- However And(Seq(A, B), C) = 3, with choices {A,B,C}, {A,C,B} and {C,A,B}. It is almost the same as permutations of 3 events, but we discard the choices {B,A,C}, {B,C,A} and {C,B,A}, where B precedes A (not possible according to Seq(A, B)). To calculate we take factorial of the total number of activities in the parallel block, and divide by product of branch factorials (e.g. 3! / (2! * 1!)).

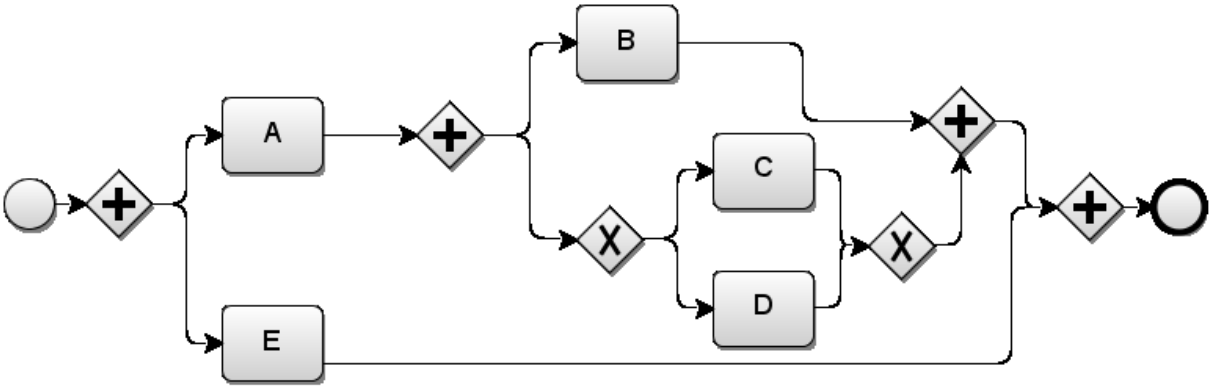

Figure 40: Process model

Figure 40 shows a model with nested blocks. First we analyse the top branch of the top-level parallel block:

1. Branch has 3 events and 4 variations
2. Branch is sequence of 1 event with 1 execution path

The formula is: $\frac{4*1}{3!*1!} * (3+1)! = \frac{4*4!}{3!} = 16$

List of all possible execution paths of the top branch is the following:

- A,B,C
- A,B,D
- A,C,B
- A,D,B

Event E could appear in the beginning, in between or in the end of the listed paths, thus the result is 4*4.

4.2.1 Process model S1

The process model is shown on Figure 33. To calculate number of distinct traces, first we consider each branch of the top parallel block separately:

1. Top branch is a sequence of 3 activities, 1 possible execution path
2. Bottom branch is a sequence of 3 activities with 1 path

Next, we calculate possible combinations for the XOR block: Xor(C, D) = 2.

Finally, the resulting is a maximum of 2 blocks: $Max\left(\frac{1*1}{3!*3!} * (3+3)!, 2\right) = \frac{6!}{3!*3!} = 20$

The result is 20, meaning that there are 20 distinct ways the process model could be executed. First, we generate a log with N randomly selected traces, where N ranges from 2 to 30. Such process is repeated 1000 times for each value of N, meaning that we generate 1000 random logs with 2 traces, then with 3 traces, etc., up to N traces. For each N the log completeness is the average of the completeness of the corresponding 1000 random logs. The completeness of each log is evaluated as mentioned before.

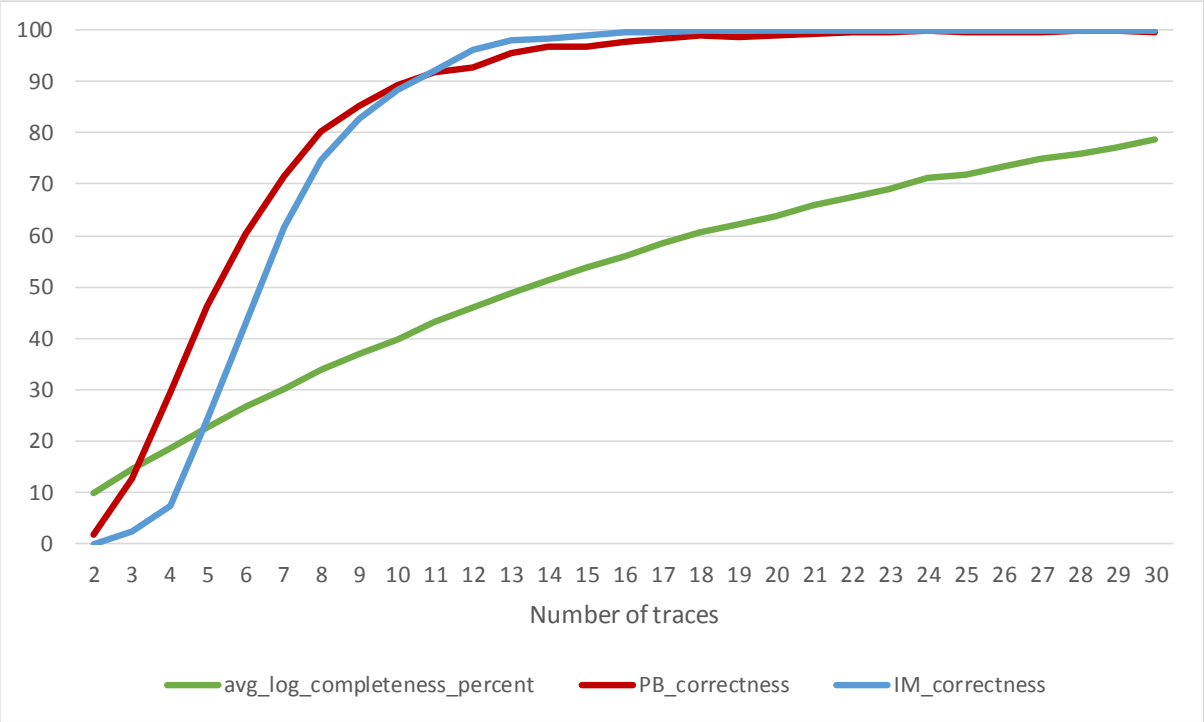

Figure 41: Graph S1

To evaluate process discovery effectiveness, we mine each log with our algorithm and compare the discovered process tree with the original process model. Correctness is calculated as the ratio between correctly discovered process trees and the total number of trees mined. In addition, we use the Inductive Miner Incompleteness (IMin) as a baseline.

In Figure 41 we see that in the beginning, while the average log completeness is below 40%, PB-Miner produces better results, than IM. After reaching 40% of the average log completeness, both graphs flatten with a discovery percent close to 100%.

Table 5 shows more detailed information about the first 10 values for number of traces (from 2 to 11, step 1) of the process model S1. We are showing in detail only these points because the results are different only in the first part of the graph. The table contains:

- Traces per log

- Average log completeness
- Correctness
- Execution time, i.e., the time needed for discovering a process tree from a log.
- Correctness comparison, i.e., a truth table for comparing discovered process tree of PB-Miner and Inductive Miner for 1000 logs.

For example, 2nd row (3 traces per log) contains the following truth table:

IM \ PB	TRUE	FALSE
TRUE	2	20
FALSE	125	853

It means that out of 1000 logs, PB and IM correctly discovered 2 process trees (TRUE, TRUE) from the same logs. PB-Miner had correctly reconstructed 125 logs (TRUE, FALSE), which IM did not reconstruct. In addition, there are 20 logs (FALSE, TRUE) which IM recognized, but PB did not. Finally, there are 853 logs (FALSE, FALSE) which were not correctly discovered by any of the miners.

Table 5: S1

Traces per log	Avg. log completeness, %	Correctness, %		Avg. execution time, ms		Correctness comparison		
		PB	IM	PB	IM	IM \ PB	TRUE	FALSE
2	9.9	1.7	0.0	3	2	TRUE FALSE	0 17	0 983
3	14.5	12.7	2.2	2	2	TRUE FALSE	2 125	20 853
4	18.7	29.4	7.2	2	2	TRUE FALSE	33 261	39 667
5	22.7	46.4	24.1	3	3	TRUE FALSE	157 307	84 452
6	26.6	60.2	42.9	3	3	TRUE FALSE	306 296	123 275
7	30.0	71.6	61.6	3	3	TRUE FALSE	493 223	123 161
8	33.7	80.3	74.5	3	3	TRUE FALSE	641 162	104 93
9	37.1	85.1	82.7	3	3	TRUE FALSE	733 118	94 55
10	39.8	89.4	88.4	3	3	TRUE FALSE	814 80	70 36
11	43.3	91.7	92.0	4	3	TRUE FALSE	851 66	69 14

4.2.2 Process model S2

The process model is shown on Figure 37. To calculate the distinct traces, we take the maximum number of distinct paths of the 2 top-level blocks:

First block is a parallel block with 2 branches:

1. 3 events, 2 variations
2. 4 events, 2 variations

Second block is a parallel block with 3 branches:

1. 2 events, 2 variations
2. 3 events, 1 path
3. 3 events, 2 variations

Thus the formula is: $Max(\frac{(2*2)}{3!*4!} * (3+4)!, \frac{2*1*2}{2!*3!*3!} * (2+3+3)!) = Max(\frac{4*7!}{3!*4!}, \frac{4*8!}{2!*3!*3!}) = 2240$

For this and the rest process models we generated 100 random logs for each trace step.

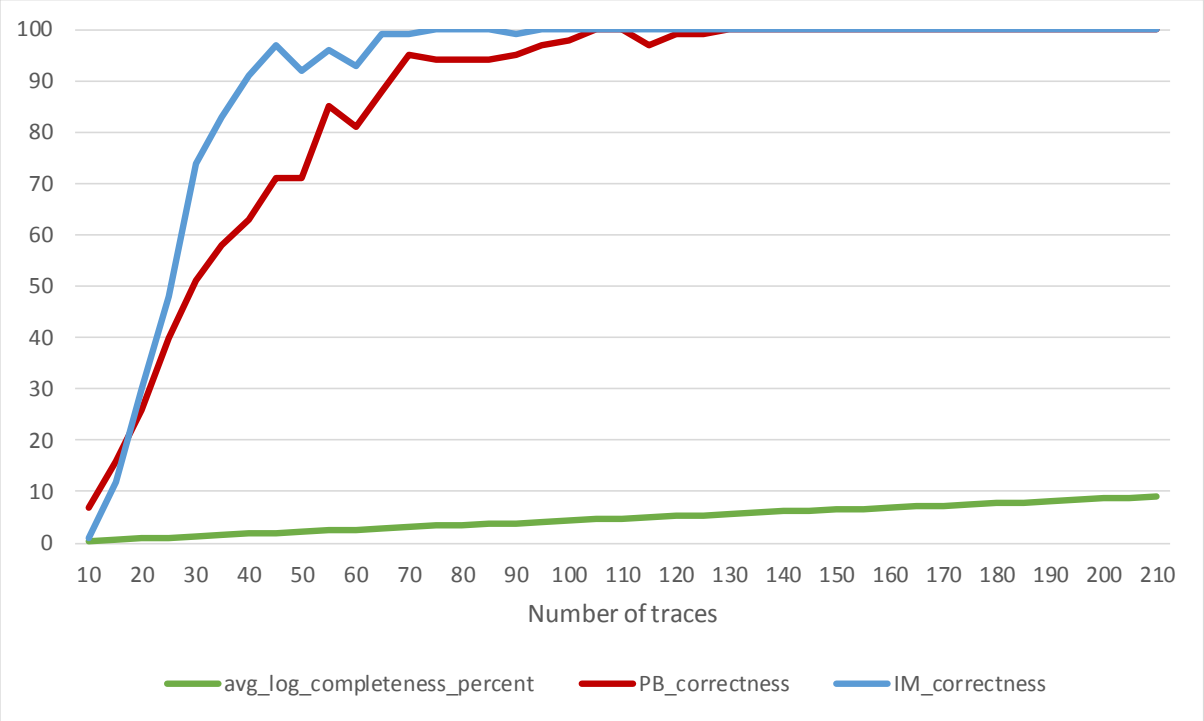

Figure 42: Graph S2

In Figure 42 we see that in the beginning, while the average log completeness is below 4.5%, Inductive Miner produces better results, than PB-Miner. After reaching 4.5% of the average log completeness, both graphs flatten with a discovery percent close to 100%.

Table 6 shows more detailed information about the first 10 values for number of traces (from 10 to 100, step 10) of the process model S2.

Table 6: S2

Traces per log	**Avg. log completeness, %**	**Equal trees, %**		**Avg. execution time, ms**		**Correctness comparison**		
		PB	IM	PB	IM	IM \ PB	TRUE	FALSE
10	0.4	7.0	1.0	85	83	TRUE FALSE	0 7	1 92
20	0.9	26.0	30.0	41	49	TRUE FALSE	6 20	24 50
30	1.3	51.0	74.0	51	48	TRUE FALSE	38 13	36 13

40	1.8	63.0	91.0	61	47	TRUE FALSE	58 5	33 4
50	2.2	71.0	92.0	70	44	TRUE FALSE	67 4	25 4
60	2.6	81.0	93.0	84	37	TRUE FALSE	74 7	19 0
70	3.1	95.0	99.0	73	28	TRUE FALSE	94 1	5 0
80	3.5	94.0	100	55	17	TRUE FALSE	94 0	6 0
90	3.9	95.0	99.0	66	19	TRUE FALSE	94 1	5 0
100	4.4	98.0	100	65	19	TRUE FALSE	98 0	2 0

4.2.3 Process model S3

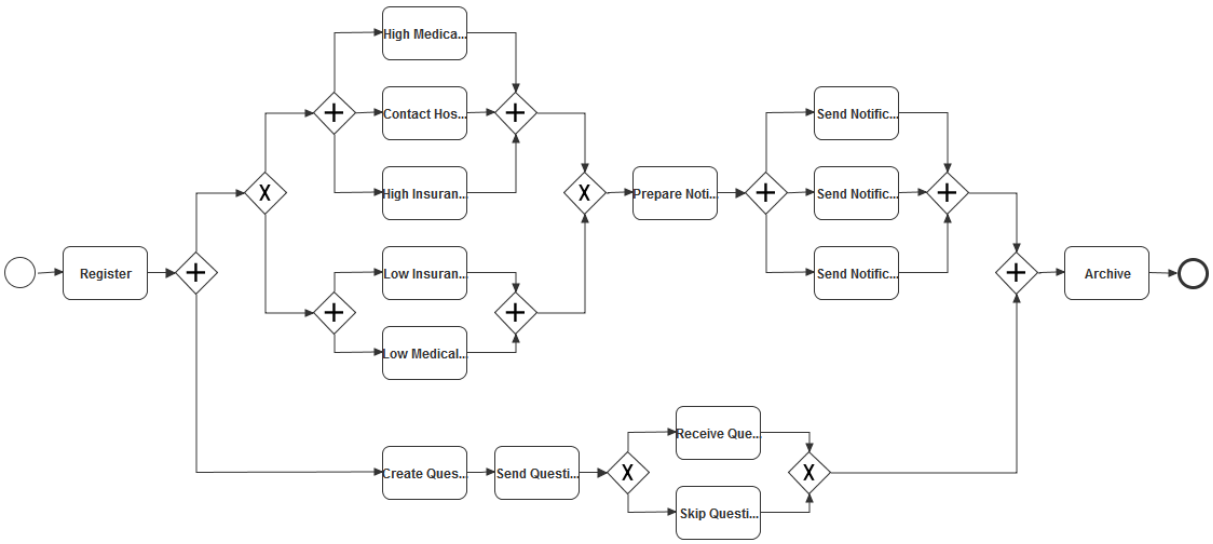

Figure 43: Process model S3

Process model S3 shown in Figure 43, is an artificial mode which corresponds to the handling of health insurance claims in a travel agency.

To compute the number of distinct traces, we analyse the top-level parallel block:

- Bottom branch has 3 events, 2 variations.
- Top branch has either 7 events and 3!*3! variations, or 6 events and 2!*3! variations.

Thus the formula is: $\frac{2*3!*3!}{3!*7!} * (3+7)! + \frac{2*2!*3!}{3!*6!} * (3+6)! = \frac{12*10!}{7!} + \frac{4*9!}{6!} = 10656$

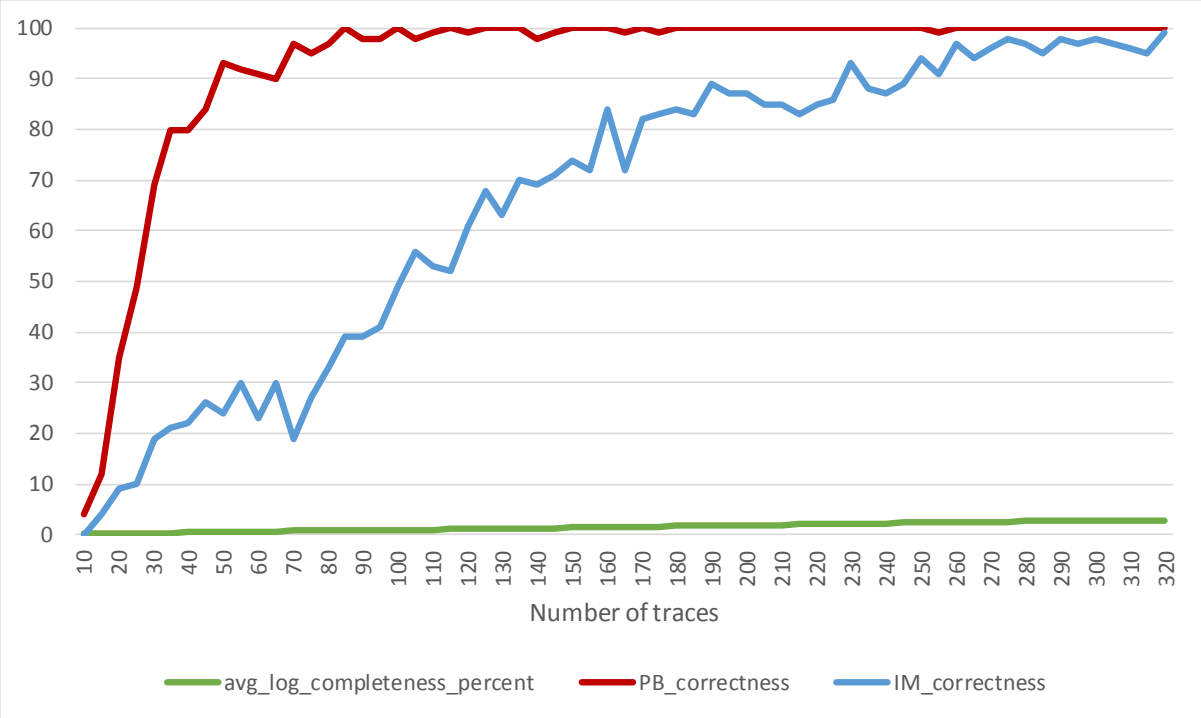

Figure 44: Graph S3

In Figure 44 we see that in the beginning, while the average log completeness is below 3%, PB-Miner produces better results, than Inductive Miner. The reason behind such difference could be explained with branch interference of the top-level parallel block. Top branch has several nested structured, which does not allow Inductive Miner to properly identify branches.

Table 7 shows more detailed information about the first 10 values for number of traces (from 10 to 100, step 10) of the process model S3.

Table 7: S3

Traces per log	**Avg. log completeness, %**	**Equal trees, %**		**Avg. execution time, ms**		**Correctness comparison**		
		PB	IM	PB	IM	IM \ PB	TRUE	FALSE
10	0.1	4.0	0.0	108	90	TRUE FALSE	0 4	0 96
20	0.2	35.0	9.0	31	60	TRUE FALSE	5 30	4 61
30	0.3	69.0	19.0	28	49	TRUE FALSE	15 54	4 27
40	0.4	80.0	22.0	30	46	TRUE FALSE	21 59	1 19
50	0.5	93.0	24.0	37	50	TRUE FALSE	23 70	1 6
60	0.6	91.0	23.0	41	47	TRUE FALSE	23 68	0 9
70	0.7	97.0	19.0	45	48	TRUE FALSE	19 78	0 3

80	0.7	97.0	33.0	51	50	TRUE FALSE	31 66	2 1
90	0.8	98.0	39.0	59	52	TRUE FALSE	38 60	1 1
100	0.9	100	49.0	63	53	TRUE FALSE	49 51	0 0

4.2.4 Process model S4

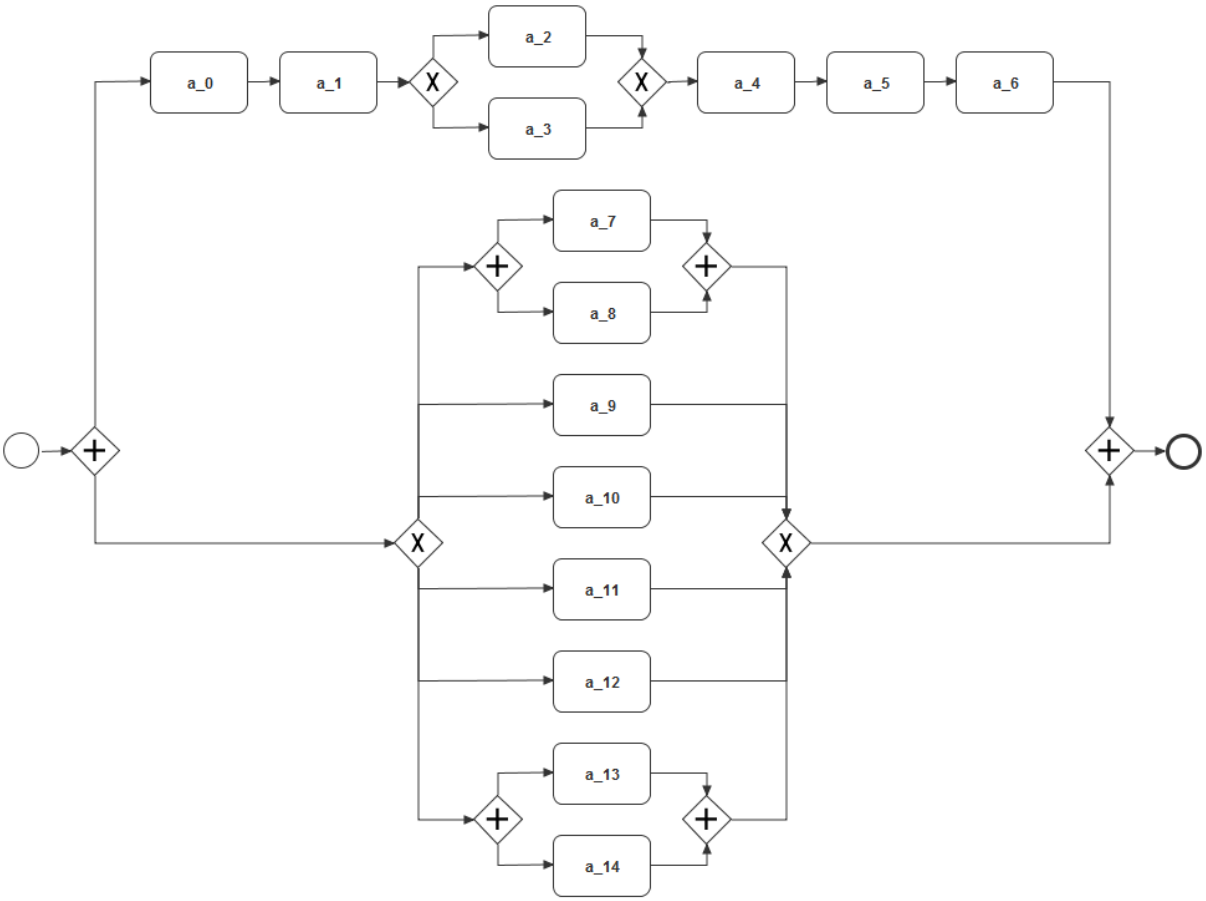

Figure 45: Process model S4

To calculate the distinct traces, we analyse each branch of the top-level parallel block:

1. Top branch - 6 events, 2 variations.
2. Bottom branch – could be 2 events of 4 variations, or sequence of 1 event with 4 variants.

Thus the formula is: $\frac{(2*4)}{6!*2!} * (6+2)! + \frac{2*4}{6!*1!} * (6+1)! = \frac{4*8!}{6!} + \frac{8*7!}{6!} = 4*7*8 + 7*8 = 280$

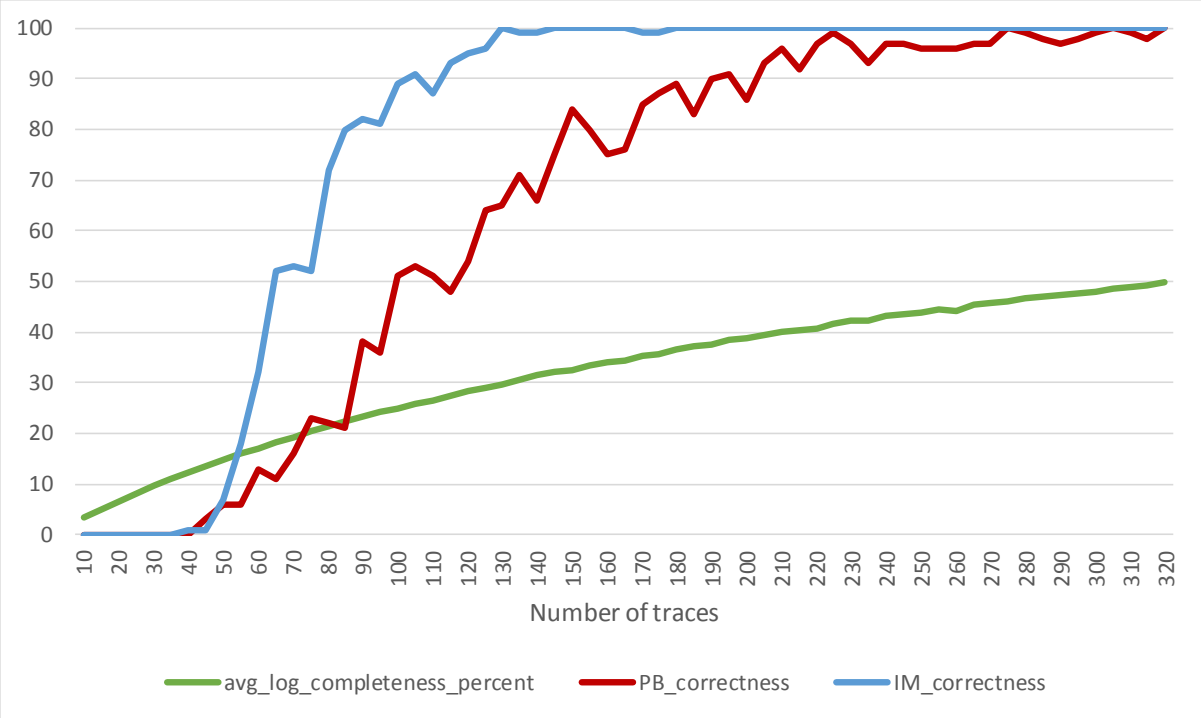

Figure 46: Graph S4

In Figure 46 both miners are less effective with dealing with incompleteness, since they are not able to discover a correct process tree for completeness values lower than 14%. One explanation for this behaviour is that the process model contains several start events. To discover the process model correctly they should appear at the first position in a trace. This is also an explanation why the PB-Miner is less effective than Inductive Miner in this case.

Table 8 shows more detailed information about the first 10 values for number of traces (from 10 to 100, step 10) of the process model S4.

Table 8: S4

Traces per log	**Avg. log completeness, %**	**Equal trees, %**		**Avg. execution time, ms**		**Correctness comparison**		
		PB	IM	PB	IM	IM \ PB	TRUE	FALSE
10	3.4	0.0	0.0	188	153	TRUE FALSE	0 0	0 100
20	6.7	0.0	0.0	125	212	TRUE FALSE	0 0	0 100
30	9.7	0.0	0.0	63	148	TRUE FALSE	0 0	0 100
40	12.2	0.0	1.0	66	136	TRUE FALSE	0 0	1 99
50	14.7	6.0	7.0	60	124	TRUE FALSE	0 6	7 87
60	17.0	13.0	32.0	49	119	TRUE FALSE	4 9	28 59
70	19.2	16.0	53.0	51	112	TRUE FALSE	11 5	42 42

80	21.4	22.0	72.0	49	120	TRUE FALSE	16 6	56 22
90	23.2	38.0	82.0	48	128	TRUE FALSE	32 6	50 12
100	25.0	51.0	89.0	45	132	TRUE FALSE	47 4	42 7

4.2.5 Process model S5

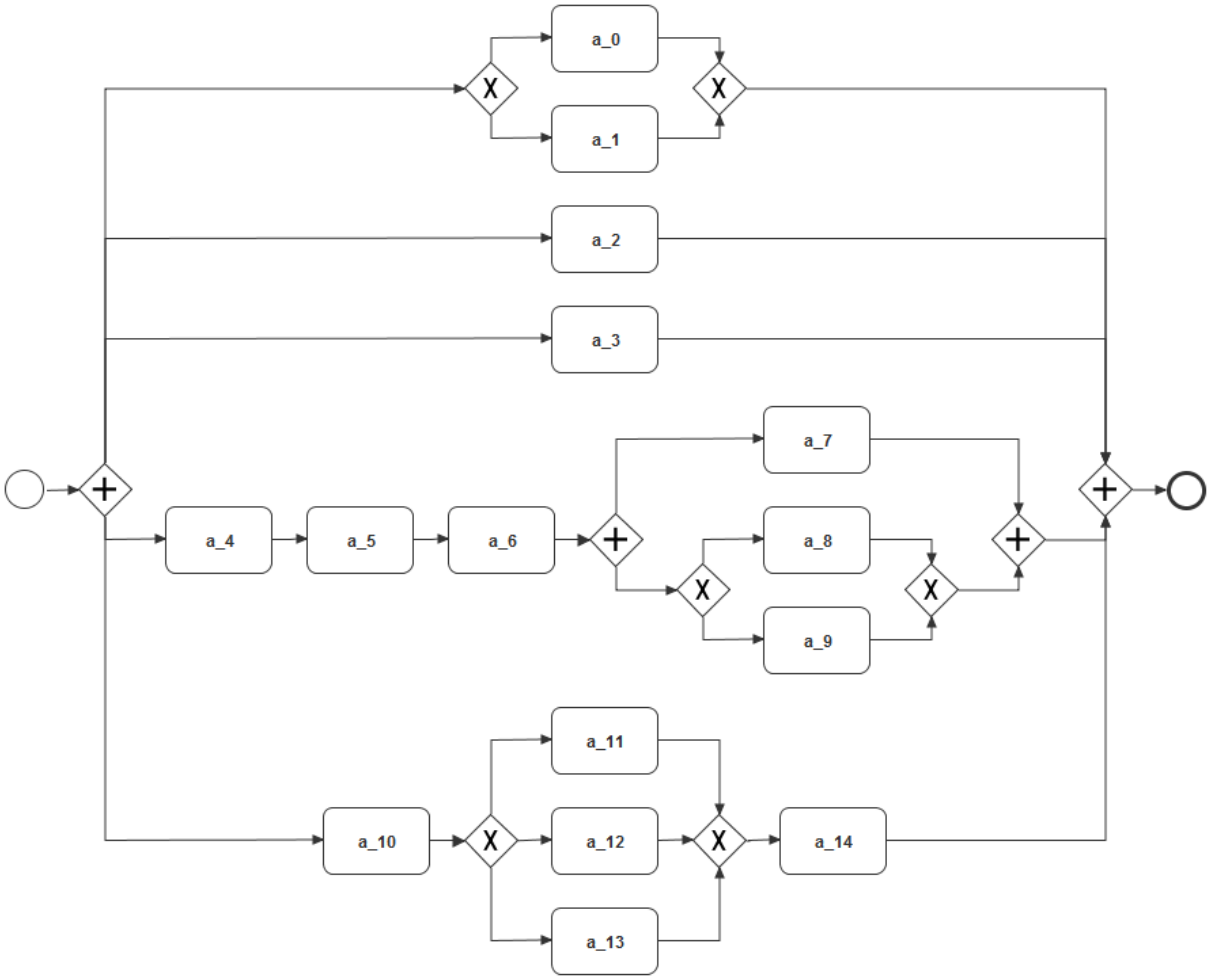

Figure 47: Process model S5

To calculate the distinct traces, we analyse each branch of the top-level parallel block (from top to bottom):

3. 1 event in branch, 2 variations
4. 1 event, 1 path
5. 1 event, 1 path
6. 5 event sequence, 4 variations
7. 3 event sequence, 3 variations

Thus the formula is: $\frac{(2*1*1*4*3)}{1!*1!*1!*5!*3!} * (1+1+1+5+3)! = \frac{4!*11!}{5!*3!} = 1330560$

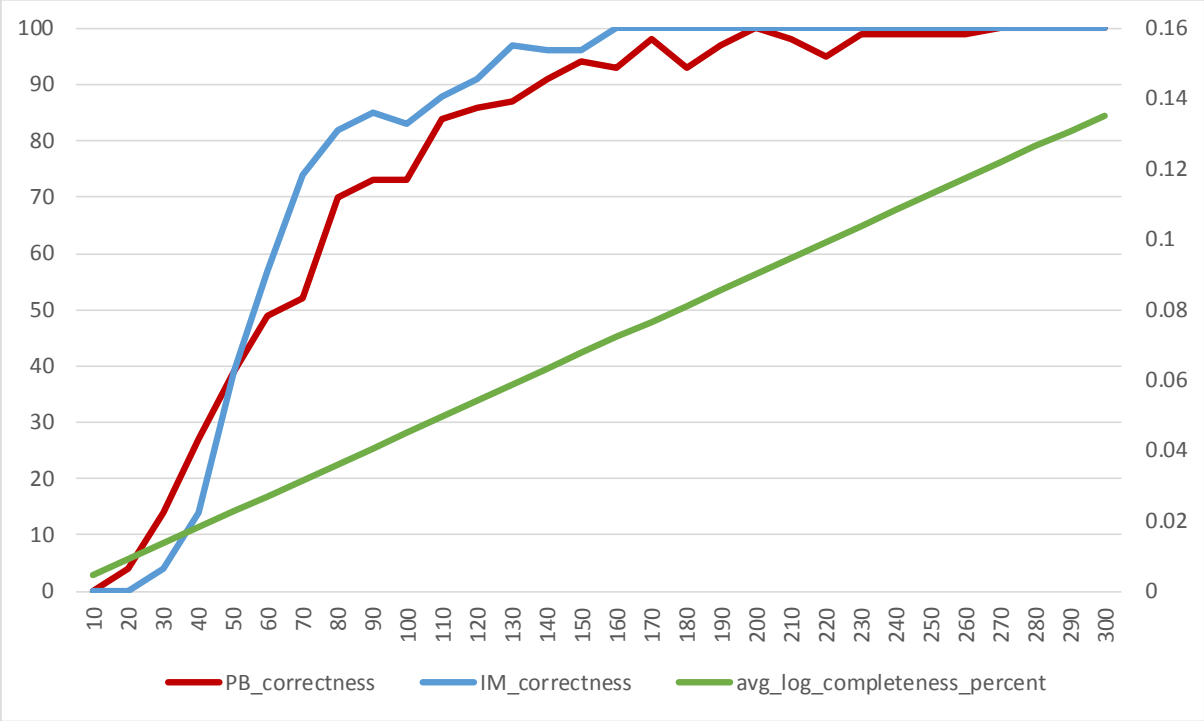

Figure 48: Graph S5

From Figure 48 it is clear that both miners can deal with incompleteness (both of them produce perfect results with 250 traces).

Table 9 shows more detailed information about the first 10 values for number of traces (from 10 to 100, step 10) of the process model S5.

Table 9: S5

Traces per log	**Avg. log completeness, %**	**Equal trees, %**		**Avg. execution time, ms**		**Correctness comparison**		
		PB	IM	PB	IM	IM \ PB	TRUE	FALSE
10	0.00450938	0.0	0.0	169	181	TRUE FALSE	0 0	0 100
20	0.009018759	4.0	0.0	64	153	TRUE FALSE	0 4	0 96
30	0.013528139	14.0	4.0	73	159	TRUE FALSE	1 13	3 83
40	0.018037518	27.0	14.0	23	183	TRUE FALSE	3 24	11 62
50	0.022542388	39.0	39.0	61	201	TRUE FALSE	11 28	28 33
60	0.027056277	49.0	57.0	30	223	TRUE FALSE	29 20	28 23
70	0.031565657	52.0	74.0	44	199	TRUE FALSE	39 13	35 13
80	0.036075036	70.0	82.0	38	187	TRUE FALSE	56 14	26 4

90	0.040579906	73.0	85.0	42	161	TRUE FALSE	66 7	19 8
100	0.045093795	73.0	83.0	46	138	TRUE FALSE	62 11	21 6

4.2.6 Process model S6

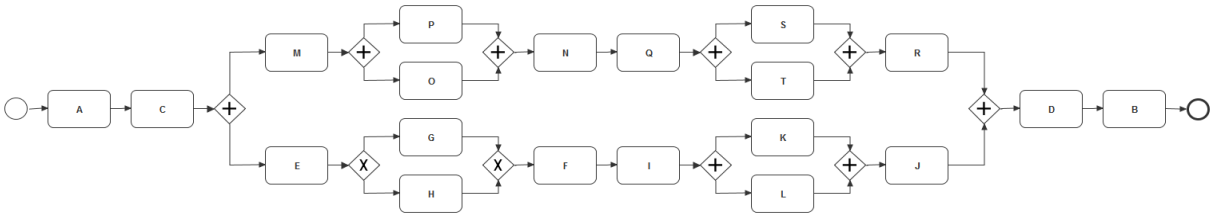

Figure 49: Process model S6

To calculate number of distinct traces, we consider each branch of the top-level parallel block separately:

1. Top branch - 8 events, 4 distinct paths
2. Bottom branch – 7 events, 4 distinct paths

Thus, the resulting formula is: $\frac{4}{8!} * \frac{4}{7!} * (8+7)! = \frac{4*4*15!}{7!*8!} = 102960$

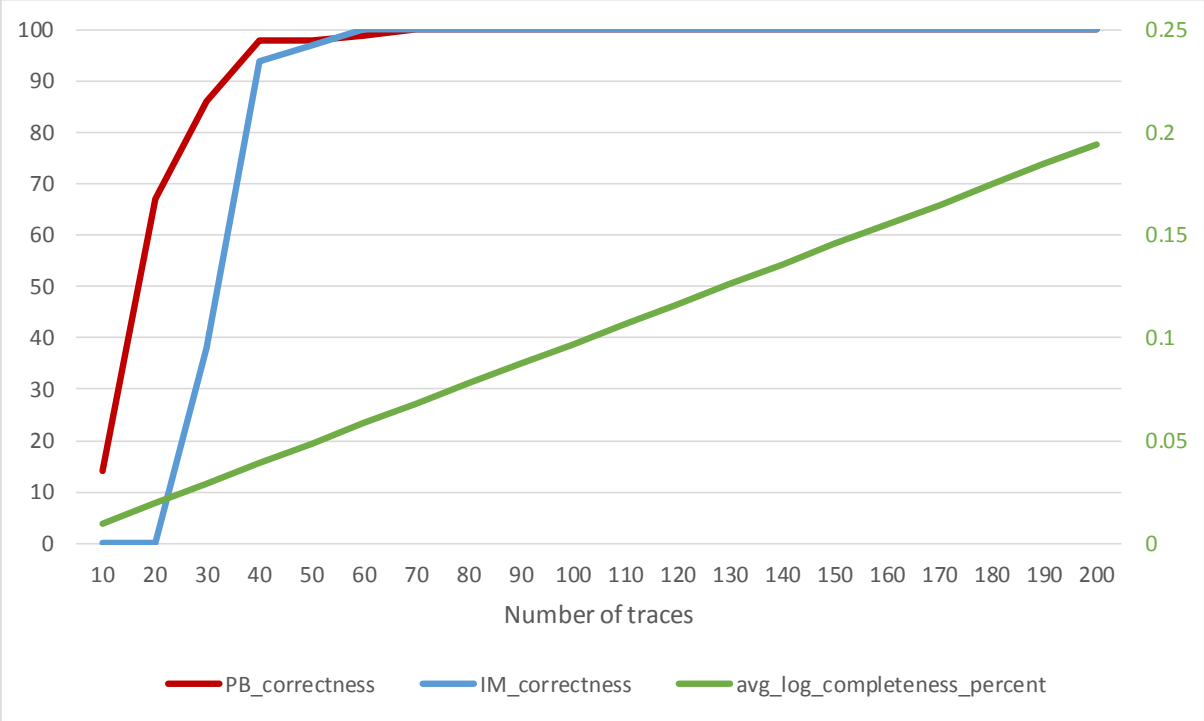

Figure 50: Graph S6

Both plots in Figure 48 and Figure 50 show that that for a very high degree of incompleteness both miners can discover the correct process tree (a completeness value corresponding to 0.07 and 0.1 respectively).

Table 10 shows more detailed information about the first 10 values for number of traces (from 10 to 100, step 10) of the process model S6.

Table 10: S6

Traces per log	Avg. log completeness, %	Equal trees, %		Avg. execution time, ms		Correctness comparison		
		PB	IM	PB	IM	IM \ PB	TRUE	FALSE
10	0.00971251	14.0	0.0	61	469	TRUE FALSE	0 14	0 86
20	0.019425019	67.0	0.0	23	576	TRUE FALSE	0 67	0 33
30	0.029137529	86.0	38.0	26	595	TRUE FALSE	33 53	5 9
40	0.038830614	98.0	94.0	26	425	TRUE FALSE	92 6	2 0
50	0.048543124	98.0	97.0	32	284	TRUE FALSE	95 3	2 0
60	0.058255633	99.0	100	38	228	TRUE FALSE	99 0	1 0
70	0.067948718	100	100	43	218	TRUE FALSE	100 0	0 0
80	0.07767094	100	100	50	208	TRUE FALSE	100 0	0 0
90	0.087373737	100	100	55	205	TRUE FALSE	100 0	0 0
100	0.09705711	100	100	62	201	TRUE FALSE	100 0	0 0

4.3 Performance analysis

To evaluate the efficiency we measured the process discovery time for all 6 process models, increasing the number of traces from 100 to 10000 (11 marks, step 1000) in a log. For every step we generated 100 random logs per model, and calculated the average execution time.

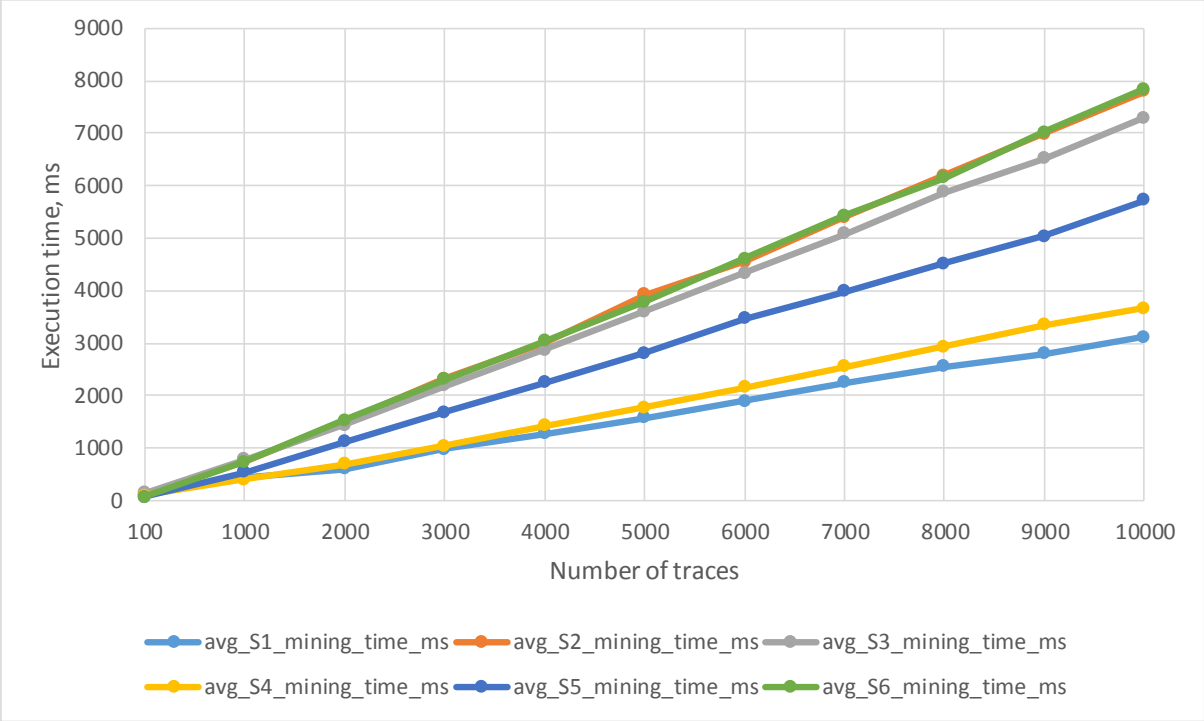

Figure 51: Performance graph

The result is shown in Figure 51, where horizontal axis is number of traces and vertical is the average mining time in milliseconds. For exact values of performance measurement results see Performance analysis dataset. From the graphs we conclude that the algorithm execution time is linear, and directly proportional to the number of traces in a log. The slope angle positively correlates with the process model complexity.

5 Conclusions

In this work, we introduced a process discovery algorithm for dealing with complex parallel behaviour. First, the algorithm discovers group of events belonging to a parallel block, then it determines the events belonging to each branch of the block. Finally, the algorithm iteratively replaces every branch of every parallel block with a placeholder event, which allows us to discover a correct process structure. Such a divide and conquer approach allows us to recursively apply the algorithm, and reconstruct a process model in a hierarchy.

To evaluate mining abilities of our algorithm we performed several qualitative and quantitative tests. The effectiveness of the mining algorithm was compared to existing process discovery algorithms (the Inductive Miner incompleteness variant (IMin) was used as a comparison baseline) using an automated test suite on randomly generated logs of 6 artificial process models. The results showed that on average our algorithm has an effectiveness that is comparable with the one of IMin. However, for logs containing a small amount of traces (less than 20), the proposed approach always performed better than the baseline. In addition, quantitative analysis showed numerous cases, for each process model, where our approach produced a correct process model, whereas IMin did not. Efficiency tests on randomly generated logs showed a linear dependency of the execution time with respect to the number of traces in the log.

5.1 Future work

There are several directions for future work:

- loops;
- more precise mapping to branches of unassigned events;
- qualitative comparison of discovery algorithms which work with incompleteness;

The algorithm described in this work is novel, and implemented as proof of concept, thus has some limitations which could be lifted as future work. Using advanced loop unrolling techniques, or a combination of several simple filtering techniques, could allow the algorithm to work with loops and duplicate events. For example, loop processing algorithms could pre-process a log, and feed a placeholder log without loops to our algorithm.

Unassigned events of a parallel block are not guaranteed to be assigned to a correct branch. We use Heuristics Miner, as it reflects the most common path, however incomplete logs might have skewed common paths, and thus other more precise techniques could be applied. One of the possible improvements would be directed (guided) assignment with the help of whitelist and blacklist of possible branches. Another idea is to use association rule mining (for example Apriori algorithm) to find frequent and interdependent group of events according to a predefined support threshold.

In our work, we calculated log completeness ratio as a number of distinct traces observed in a log, divided by the total number of possible permutations. However, this metric is very generic and does not provide qualitative information about the distinct traces. For example, having N traces which reflect variations only in the second half of the original model, makes it impossible to correctly detect the first half of the model. Furthermore, by estimating log incompleteness qualitatively, we would be able to qualitatively compare discovery algorithms to understand what characteristics of a process model can be discovered more easily with one algorithm or another.

Bibliography

[1] "Process Mining Manifesto," in *Business Process Management Workshops*, Springer, 2012, pp. 169-194.

[2] W. v. d. Aalst, Process Mining - Discovery, Conformance and Enhancement of Business Processes, Eindhoven: Springer, 2011.

[3] W. v. d. Aalst, T. Weijters and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Transactions on Knowledge and Data Engineering - TKDE,* vol. 16, no. 9, pp. 1128-1142, 2004.

[4] "Wikipedia: BPMN," [Online]. Available: https://en.wikipedia.org/wiki/Business_Process_Model_and_Notation. [Accessed April 2015].

[5] S. J. Leemans, D. Fahland and W. v. d. Aalst, "Discovering Block-Structured Process Models from Incomplete Event Logs," Eindhoven, 2014.

[6] D. Schunselaar, H. Verbeek, B. v. Dongen and S. Leemans, "Process Tree Package," Eindhoven, 2014.

[7] C. W. Günther and E. Verbeek, "OpenXES Developer Guide v2.0," Eindhoven, 2014.

[8] W. v. d. Aalst, "Process Mining tools," 2011. [Online]. Available: http://www.processmining.org/logs/start. [Accessed April 2015].

[9] C. W. Günther and E. Verbeek, "XES Standard Definition v2.0," Eindhoven, 2014.

[10] S. J. Leemans, D. Fahland and W. v. d. Aalst, "Discovering block-structured process models - A Constructive Approach," Springer, Eindhoven, 2013.

[11] A. Weijters and J. Ribeiro, "Flexible Heuristics Miner," Eindhoven, 2011.

[12] H. A. Reijers, T. Slaats and C. Stahl, "Declarative modeling – An academic dream or the future for BPM?," in *BPM*, 2013.

[13] F. M. Maggi, T. Slaats and H. A. Reijers, "The Automated Discovery of Hybrid Processes," in *BPM*, 2014.

[14] J. D. Smedt, S. v. d. Broucke, J. D. Weerdt and J. Vanthienen, "A Full R/I-net Construct Lexicon for Declare Constraints," Leuven, 2015.

[15] G. D. Giacomo, M. Dumas, F. M. Maggi and M. Montali, "Declarative Process Modeling in BPMN," 2015.

[16] R. Conforti, M. Dumas, L. Garcıa-Banuelos and M. L. Rosa, “Beyond Tasks and Gateways: discovering BPMN models with subprocesses, boundary events and activity markers,” in *BPM*, 2014.

[17] M. Dumas, M. L. Rosa, J. Mendling and H. A. Reijers, Fundamentals of Business Process Management, Springer, 2013.

[18] W. v. d. Aalst, T. Weijters and L. Maruster, “Workflow mining: Discovering process models,” *IEEE Transactions on Knowledge and Data Engineering - TKDE,* vol. 16, no. 9, pp. 1128-1142, 2004.

Appendices

I. Performance analysis dataset

The following table shows detailed information of performance analysis results.

Traces	Avg. model mining time in ms, using 1000 samples (random logs)					
	S1	S2	S3	S4	S5	S6
100	111	99	139	101	61	70
1000	443	749	786	405	536	744
2000	598	1508	1447	693	1127	1525
3000	988	2319	2182	1051	1684	2316
4000	1274	3006	2879	1425	2253	3049
5000	1582	3931	3609	1771	2818	3794
6000	1896	4555	4336	2156	3466	4614
7000	2243	5407	5081	2551	3980	5431
8000	2551	6199	5882	2940	4524	6154
9000	2795	6986	6521	3344	5045	7016
10000	3112	7798	7292	3667	5719	7845

II. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Bogdan Semiletko** (date of birth: 28.08.1989),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright, of my thesis

   Dealing with Complex Parallel Structures in Process Discovery,

   supervised by **Fabrizio Maria Maggi.**

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **25.05.2015**