Scalable Discovery of Hybrid Process Models in a Cloud Computing Environment

Long Cheng, Boudewijn F. van Dongen and Wil M.P. van der Aalst *Senior Member, IEEE*

Abstract—Process descriptions are used to create products and deliver services. To lead better processes and services, the first step is to learn a process model. Process discovery is such a technique which can automatically extract process models from event logs. Although various discovery techniques have been proposed, they focus on either constructing formal models which are very powerful but complex, or creating informal models which are intuitive but lack semantics. In this work, we introduce a novel method that returns hybrid process models to bridge this gap. Moreover, to cope with today's big event logs, we propose an efficient method, called *f*-HMD, aims at scalable hybrid model discovery in a cloud computing environment. We present the detailed implementation of our approach over the Spark framework, and our experimental results demonstrate that the proposed method is efficient and scalable.

Keywords—Process discovery; hybrid process model; event log; big data; service computing; cloud computing

1 Introduction

Event data collected by modern information systems can be used to extract non-trivial knowledge and interesting insights using *process mining* techniques [1]. Specifically, *process discovery*, as one of the core tasks in process mining, can automatically extract a process model from a given event log and such a model can be used to create products and deliver services [2]. Examples include, but are not limited to the domains of telecom and insurance [3], [4].

Up to now, various discovery approaches have been proposed in the process mining domain. According to their discovered models, all the techniques can be divided into two main categories: (1) *formal models*, which explicitly and unambiguously specify all possible orderings of behavior. This model can be acquired by various discovery algorithms such as the Inductive Miner [5]; and (2) *informal models*, which provide insights using diagrams that have no formal semantics and can not be used to reason about behavioral properties.

Generally, formal models explicitly offer guidance for processes, which undoubtedly can provide value for process execution. Regardless, they are typically inclined to enforce guidelines [6], which is inflexible in terms of covering all the possible variability. Moreover, this kind of models has strong semantics that govern the overall interplay of the actions and objects of a process. This makes discovery challenging, and the resulting models could be also complex and hard to understand for non-professional users. In comparison, informal models allow users to describe a process through simple "boxes and arcs", which makes the discovery process much easier and discovered models more intuitive. However, informal models do not have clear semantics which are actually important for operational processes in practice.

Considering the advantages of the two models above, it makes sense to try and combine the best of both worlds in the form of a hybrid process model. Such models only formalize the parts that are clear and supported by the data. Things that are less clear or not supported by enough evidence, result in informal annotations that are still useful for users. However, with the spectacular growth of event data, it will be very challenging to discover such a hybrid model in a scalable way.

To handle the problem, efficient hybrid model discovery over cloud-based platforms such as MapReduce [7] and Spark [8] becomes desirable. The reason is that these platforms can efficiently extend the amount of available computing resources to provide a straightfoward scale-out capability [9]. Moreover, these platforms have integrated parallelization, fault tolerance and load balancing in a simple programming framework, and thus allow their implementations for a easy deployment in cloud.

In this paper, we focus on efficient hybrid process model discovery in a cloud computing environment. To achieve this goal, we propose an approach, called *f*-HMD (*f*ilter-based **H**ybrid process **M**odel **D**iscovery). On a reference cloud-based implementation, we experimental demonstrate the effectiveness and efficiency of our method. The main contributions of this paper can be summarized as follows:

- We introduce a hybrid process model and propose a method (HMD) to discover such a model over big event logs in a cloud computing environment.
- We analyze the performance issues of HMD and

- *L. Cheng is with the School of Computer Science, University College Dublin, Dublin 4, Ireland. E-mail: long.cheng@ucd.ie*
- *B.F.v. Dongen is with the Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven 5600MB, The Netherlands. E-mail: B.F.v.Dongen@tue.nl*
- *W.v.d. Aalst is with the Lehrstuhl für Informatik 9 / Process and Data Science, RWTH Aachen University, D-52056 Aachen, Germany. E-mail: wvdaalst@pads.rwth-aachen.de*

online available at https://doi.org/10.1109/TSC.2019.2906203

propose two strategies to speed up its discovery process. Specifically, we incorporate two lightweight filters in HMD (i.e., 2f-HMD) to reduce its computation workloads.
- We present the detailed implementation of our approach over the Spark framework [8]. As filtering may change the final output, we discuss about the safety of our filters and theoretically provide the safe ranges for implementation parameters.
- We conduct a detailed experimental evaluation of our approach and the results demonstrate our approach is efficient and scalable. Specifically, it can process 2.52 million traces having about 100 million events in about 1 minute over 16 nodes (64 cores).

The remainder of this paper is organized as follows. Section 2 gives a brief introduction about related concepts in process discovery. Section 3 introduces the detailed design of our approach. Section 4 presents our implementation in details. We report the experimental results in Section 5 and discuss the related work in Section 6. We finally conclude this paper in Section 7.

2 Preliminaries

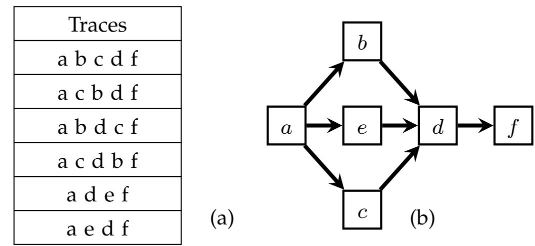
In this section, we briefly describe some of the basic concepts in process discovery. For more formal definitions and examples, we refer to the book [1].

2.1 Event Log

An *event log* is a collection of traces (process instances), where every trace is a temporally ordered sequence of events, and each event is represented by a set of attribute values (e.g., name, resource and timestamp, etc.). For example, Figure 1(a) shows an event log, which is composed of six distinct traces. Moreover, in each trace, each event is represented by its activity name. Other attributes such as time, resource, etc. have been abstracted away. In total, the log contains $5\times4+4\times2=28$ events, and the number of unique activities is 6. For this condition, the log is also referred to as an *activity log*, which is an abstraction of the event logs as found in practice [1]. In the remainder of this paper, without sacrificing generality, we focus on discovering a hybrid model from such an activity log.

2.2 DFG

An informal process model is always represented in the form of a *directly-follows graph* (DFG), which can be derived from a log and describes what activities follow one another directly, and with which activities a trace starts or ends. In a DFG, there is an arc from an activity a to an activity b if a is followed directly by b, and the weight of the arc denotes how often that happened. In practice, based on either domain knowledge or statistical information (e.g., frequency), some arcs could be removed from a DFG.



Traces
a b c d f
a c b d f
a b d c f
a c d b f
a d e f
a e d f

Fig. 1: An event log and informal model (we ignore the initial and end notations in the model for simplification).

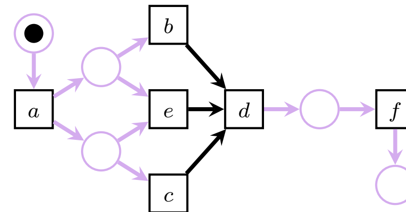


Fig. 2: An example of hybrid process model.

Because a DFG can represent processes in a direct, simple and visual way, it has become the mainstream model in current process mining products. Commercial process mining tools such as Disco [10], Celonis [11], ProcessGold [12] and Minit [13] often generate informal models based on such DFGs, and most of them produce DFGs where infrequent activities and arcs are removed using sliders. As DFGs do not contain any execution semantics and have difficulties dealing with concurrency, we consider DFGs only as an intermediate result, not suitable to show the real process logic. In this paper, we construct a DFG from an event log *without* removing any of the discovered arcs, and show that this can be done in a parallel manner in a cloud computing environment.

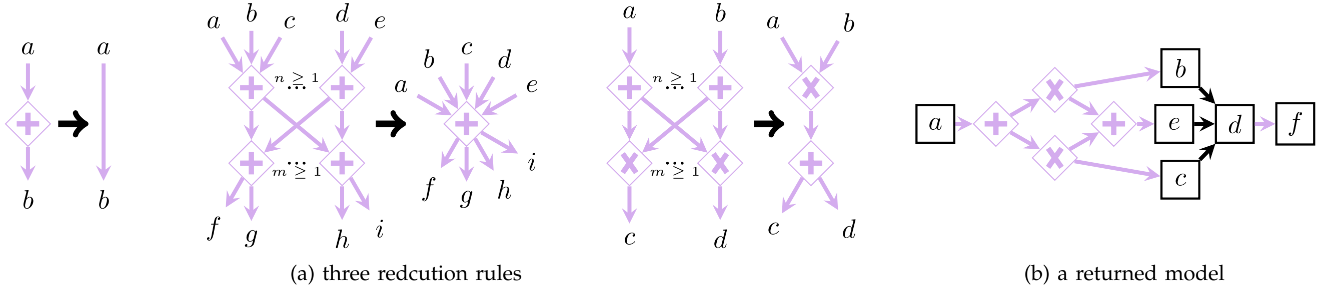
2.3 Petri net

A formal process model can be modeled in terms of *Petri nets* [14], using three types of elements: *places*, *transitions*, and *arcs*. We have incorporated an uncompleted Petri net in a hybrid model as demonstrated in Figure 2. There, each activity represents a transition, each circle represents a place, and places and transitions are connected by arcs (having *purple*/gray color).

Compared to an informal model, a formal one can provide more explicit information on the possible process executions. For instance, we only know that the activities $\{b, c, e\}$ follow a in some traces from the DFG in Figure 1(b), but it remains unclear what can happen after a in a process. In contrast, using semantics from the Petri net as shown in Figure 2, we know that once the transition a is fired, two tokens will be produced: one for each output place. After a either e or both b and c happen. The ordering of b and c is not fixed.

2.4 BPMN

Business Process Model and Notation (BPMN) is a rich language that provides modelers with a large collection



(a) three redcution rules

(b) a returned model

Fig. 3: Some reduction rules and the final hybrid model returned by our approach.

of object types to represent various aspects of a business process, including the control-flow, data, resources and exceptions [15]. Compared to Petri nets, BPMN is specifically tailored to modeling business processes at a conceptual level, meaning that they are often used for communication between stakeholders in the processes. Therefore, we will visualize our final models using BPMN, and restrict ourselves to a small subset of BPMN having clear execution semantics.

To transform a process model from Petri nets to BPMN, a typical way is using logical gateways to replace the *places*. There are three rules for this transformation: (1) every transition/activity with input places has an *and-join* gateway connected with precise arcs; (2) every transition with output places has an *and-split* gateway connected with precise arcs; and (3) all places are replaced by *xor*-gateways. Moreover, six reduction rules can be applied to simplify the replaced model by reducing its gateways, and Figure 3(a) shows three of them: (1) a gateway can be removed if it has only one input a and one output b; (2) two sets of fully connected AND gateways can be simplified when there are only inputs for one set and outputs for the other; and (3) for the case that a set of AND gateways are fully connected with a set of XOR gateways, if there is only one input for each AND and only one output for each XOR, then the gateways can be simplified. These three rules also work when replacing the AND gateways with XOR while replacing the XOR gateways with AND (i.e., another three rules). After these replacement and simplification operations, the model in Figure 1(b) can be transformed into the hybrid model as demonstrated in Figure 3(b).

3 Our Approach

In this section, we present the details of our approach and show how we realize the approach in a cloud computing enviroment (e.g., over Spark [8]).

3.1 Overview

As we have described, end-users like to use informal process models, and it is unlikely that this will change. Moreover, Petri nets (and the like) are perceived as being too complex in terms of discovering and understanding. Futhermore, current mining algorithms are often not sure about some parts of a model, and forcing explicit semantics only provides fake certainty [16]. In comparison, informal models are simple, but lack execution semantics. In fact, parts of an informal model can actually be enhanced by formalizing the parts that can be captured in precise modeling constructs with good confidence.

Our discovery technique only inserts formal dependencies between activities if they have a good quality, i.e., there is sufficient evidence in the event log to support such formalization. By replacing informal dependencies with formal ones whenever possible, we aim to construct a hybrid process model to bridge the gap between the two different models and consequently combine their advantages in a single model, i.e., a hybrid process model. Generally, our model discovery process can be divided into the following three steps:

- *Step 1*: construct a DFG from an event log.
- *Step 2*: discover *places* in the DFG by checking the feasibility of such places based on the evidence in the event log in a exhaustive way.
- *Step 3*: replace the discovered places with logical gateways from BPMN and simplify the gateways by applying reduction rules.

As an example, for the event log in Figure 1(a), we can first get a DFG as shown in Figure 1(b). Then, we examine all the possible places over the input log and extend the DFG model with discovered places. As a result, we get a model as illustrated in Figure 2. Finally, we replace the places and output a hybrid model as shown in Figure 3(b). Compared to a fuzzy model (i.e., DFG), all the precise part (arcs and gateways) in our hybrid model will be marked in a configurable color (e.g., *purple* in our case) to highlight the parts that have formal semantics. In the meantime, the rest parts (arcs) are kept as normal, indicating that there is no confident semantics between the responsible activities.

3.2 Hybrid Model Discovery

Based on the above steps, Algorithm 1 describes the implementation of our hybrid model discovery (HMD).

3.2.1 DFG Discovery

All the traces in an input log L can be easily partitioned in a distributed enviroment. For example, in a cloud with

HDFS [17], the partitioning can be done automatically when uploading the log. Specifically, the number of traces of each partition can be balanced, by explicitly assigning the number of partitions when reading the log (using Spark). It should be noticed that the partitioning is done in trace-level and thus all the directly-follows relationships are well maintained.

On the above basis, a directly-follows graph can then be computed in a single pass over the event log in parallel. As shown in Algorithm 1 lines 1-7, for each sublog L^k at each node[1], we examine the directly-follows relationships for all the neighbored activities (line 4) and record them in a local matrix F^k (i.e., line 5, increasing the counter number by 1). This statistics can be done in parallel on all the computing nodes. This is because the computing is done on each trace σ, and all the traces are distributed and independent to each other. After that, all the values in F^k at each node k are aggregated based on their indexes to construct a new matrix F. This matrix represents all the directly-follows relationships of the log (line 8). Namely, if $F_{x,y} = 0$, the activity a_x is never followed by the activity a_y, otherwise, a_x is followed by a_y at least once and thus we add an arc between the two activities in the discovered DFG.

3.2.2 Place Discovery

Our target is to add feasible places between activities for the discovered DFG to enhance its execution semantics. To get such places, we use a *generating-and-pruning* scheme, i.e., for an input DFG, we first generate all the place candidates X, and then prune the ones that do not meet standard Petri nets semantics. The details of this process are presented in Algorithm 1 lines 9-28.

Candidate generating. For a DFG, a place could appear between two or multiple activities including directly-follows relations. Therefore, a place candidate $X_j \in X$ can be represented in the form of an arc cluster (A, B), where A and B are activity sets. For generality, let B follow A, and X_j appears between them. To quickly get all such candidates, we first represent the DFG using an adjacency list. Then, we construct an invert-list based on all the possible combinations (and selections) of the neighbors of each vertex and treat each combination as a vertex. Finally, we generate all the candidates based on all the possible combinations of neighbors of each vertex in the invert-list. An example of such a processing is shown in Figure 4. There, the DFG shown in Figure 4(a) can be represented as an adjacency list $\{\langle a|b,c,e\rangle, \langle g|b,c,e\rangle\}$, and the neighbors of the two vertexes a and g are $\{b,c,e\}$. There are 7 different combinations for the three nodes, e.g., $\{b\}$, $\{b,c\}$ and $\{b,c,e\}$ etc., therefore we can construct an invert list shown as Figure 4(b), in which a and g are the neighboring nodes. Followed by that, there are 3 possible combinations for a and g, i.e., $\{a\}$, $\{g\}$ and $\{a,g\}$, thus finally we have 21 place candidates, which are listed in Figure 4(c).

Algorithm 1 Hybrid Model Discovery (HMD)

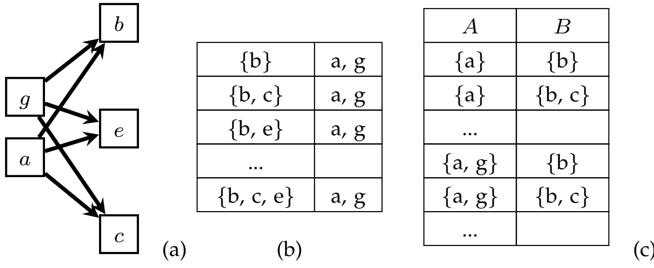
DFG discovery:
1: **for** $L^k \in L$ at each node k **parallel do**
2: Initialize a 2D matrix F^k
3: **for** $\sigma \in L^k$ **do**
4: $\forall(a_x, a_y) \in \sigma$ and $a_x \leftarrow a_y$,
5: $F^k_{x,y} += 1$ // follows relation
6: **end for**
7: **end for**
8: Combine all F^k to construct a DFG

Place discovery:
9: Generate all the place candidates X based on the discovered DFG.
10: **for** $L^k \in L$ at each node k **parallel do**
11: **for** $X_j \in X, \sigma \in L^k$ **do**
12: **if** $!(A \cap \sigma = \emptyset \wedge B \cap \sigma = \emptyset)$ **then**
13: **if** σ : *Rule 3* **then** // Rule 3
14: $D_{kj} = 1$
15: **end if**
16: **end if**
17: **end for**
18: **end for**
19: **for** $X_j \in X$ **do**
20: **if** $f(\sum_k D_{kj}) < t_3$ **then**
21: $X = X - X_j$ // pruning
22: **end if**
23: **end for**

Post-processing:
24: Partition DFG into g_1^0 and g_2 using X
25: Add places to g_1^0 and replace them with logical gateways
26: **while** $g_1^n \neq g_1^{n-1}$ **do**
27: $g_1^n = R(g_1^{n-1})$ // simplifying
28: **end while**
29: $HM = (g_1^n, g_2)$
30: Visualize HM

Candidate pruning. We prune the generated place candidates based on the semantics of Petri nets. For a candidate $X_j = (A, B)$, let T is a subset of L, and there is $T = L \setminus \{\sigma \mid A \cap \sigma = \emptyset \wedge B \cap \sigma = \emptyset, \ \forall \sigma \in L\}$. Namely, we guarantee that for each trace σ in T, there exists at least one activity in A or B that appears in the trace. Here we look for places having input set A (transitions producing a token for the place) and output set B (transitions consuming a token for the place). Using standard Petri nets semantics we need to check that: (1) we never consume a token from an empty place, and (2) at the end the place is empty. Namely, for each trace, the frequencies of transitions in A should be no less than the transitions in B at any time point. Meanwhile, the frequencies of transitions in A should match the

1. We focus on describing our algorithm at a conceptual level only. In this context, a node here refers a computing unit.



{b}	a, g
{b, c}	a, g
{b, e}	a, g
...	
{b, c, e}	a, g

A	B
{a}	{b}
{a}	{b, c}
...	
{a, g}	{b}
{a, g}	{b, c}
...	

Fig. 4: An example of candidate generation process for a given DFG.

frequencies of transitions in B at the end of the trace.

Let $\sigma(i)$ denote the first i elements of a trace σ, $|\sigma|$ be the length of the trace, $\#_v\mathcal{U}$ be the frequency of the transitions in $\mathcal{U}$ appearing in v. Then, the above two rules can be formalized as: $\forall\sigma \in T$, there is

$$\begin{cases} \#_{\sigma(i)}A \geq \#_{\sigma(i)}B, & \forall i \in [1, |\sigma|) \\ \#_{\sigma}A = \#_{\sigma}B \end{cases} \qquad \text{(Rule 3)}$$

If A and B satisfy the above rule, then a place with input transitions A and output transitions B will be added in the DFG. In our previous example in Figure 1(b), the case $A = \{a\}$ and $B = \{b, e\}$ satisfies above Rule 3, thus there is a place between them as shown in Figure 2.

Noise or infrequent behavior could occur in an event log, and such kind of data would greatly impact our pruning processing if we follow Rule 3 in a very strict way (i.e., $\forall\sigma \in T$). To handle this problem, we adopt an approximation for Rule 3. Specifically, we use a parameter γ to represent the approximation:

$$\gamma = \frac{\#\sigma,\ \sigma : \textit{Rule 3}}{|T|} \qquad \text{(Approx. 3)}$$

It means that for a place candidate (A, B), we check whether the majority of the traces in T meet the Rule 3. In our tests, we set a threshold $t_3 = 90\%$ as default, namely there will be a place between A and B if and only if $\gamma \geq t_3$. This ensures that the large majority of traces indeed complies with the place to be added.

Based on above analysis, in our algorithm, we examine each candidate X_j over the traces L^k on each node k in parallel (lines 10-18). The collected statistical information D_{kj} of Rule 3 for each candidate X_j on each node will be combined by an aggregation and the non-matched candidates will be removed if they do not meet the configured threshold t_3 (lines 19-23).

3.2.3 Post-processing

We can partition the discovered DFG into a precise part g_1 and a fuzzy part g_2 once we get the final candidates X (line 24). After adding the discovered places in g_1, based on the replacement strategies described in Section 2, we can then replace the places with logical gateways (line 25). Following by that, we repeatedly apply the simplification rules to the graph g_1 until the graph does not change any more (lines 26-28), and the final output will be constructed by the refined graph g_1 and g_2 (lines 29-30). Because a (hybrid) business process model including activities and discovered places (in the form of a graph) is typically very small, the whole post-processing will be very lightweight. Therefore, we conduct all the operations (i.e., replacement, simplification and visualization) on a single computing node (e.g., the master node in Spark). In fact, the post-processing for the two datasets we have used in our experiments in Section 5 can be always done in less than 150 milliseconds.

3.3 The f-HMD Algorithm

3.3.1 Performance Issues

In above HMD algorithm, the DFG discovery can be done by a simple statistics-based job, which would be light-weight in a distributed environment. Meanwhile, the candidate generation in the second step and post-processing in the third step can be also done in a quick way as the graph is typically small. Clearly, the process of candidate pruning in our algorithm is the most time consuming one, as we need to check all the candidates over all the traces and all their prefixes based on Rule 3.

The check operations would make the place discovery costly, especially when the number of candidates is large. In fact, this could happen even when a DFG is small, because the number could increase sharply with increasing either the size of the graph or the connections between vertexes (activities). For example, for the latest BPI Challange dataset as we used in our later evaluations (the Application log [18]), the number of generated candidates is 176798. If we have 1 million traces, then the number of trace-level examination will be around 176 billions. This will bring in great performance challenges, even when the number of computing nodes is large.

To discover a hybrid model in a reasonable time in the presence of large event logs, we propose a more efficient approach, $f$ilter-based **H**ybrid process **M**odel **D**iscovery (f-HMD), by incorporating effective strategies into the HMD algorithm. Firstly, we identify model-level constraints, to reduce the number of generated place candidates, on the basis of our observation on current methods in constructing process models. Then, we employ data-level constraints from Rule 3 and use them as light-weight filters, to prune large numbers of place candidates before the place discovery process over Rule 3. Both strategies are able to efficiently reduce the computing workloads for HMD and consequently to improve its overall performance.

3.3.2 Model-level Constraints

As described earlier, formal models can be discovered by various process discovery algorithms. We identify the model-level constraints from these approaches. Considering the amount of events in large logs as well as the quality of discovered processes (e.g., soundness and fitness), we have chosen the Inductive Miner [5] as our reference. Besides the fact that the Inductive Miner is

the state-of-the-art process discovery approach, other techniques incline to produce models that are unable to replay the log well, i.e., they either create erroneous models or have excessive runtime during executions.

The main objective of Inductive Miner is to discover a set of block-structured process models for any given logs. The discovered model can be represented as a process tree, which is a compact abstract representation of a block-structured workflow net. The root in such a tree leaves certain characteristics in the log and DFG, and the leaves are labeled with activities and all other nodes are labeled with operators [5]. The operations can characterize the exclusive choice, parallel, sequence and loop behaviors for activities of a log. For example, for exclusive choice, each trace will be generated by one child in the process tree. In fact, a workflow net is a Petri net having a single start place and a single end place, modeling the start and end state of a process. Moreover, all nodes are on a path from start to end [19]. In such scenarios, a block-structured workflow net is a hierarchical workflow net that can be divided recursively into parts having single entry and exit points[2].

Consider the discovery process in our approach, each precise part of a hybrid model is actually able to be represented as a block. Unlike the Inductive Miner, we will only need to consider the potential places between a single activity and multiple activities or reversed. This means that in our candidate generation, we only need to consider a candidate (A, B) with the cases that either the size of A or B is 1 (i.e., the number of contained transitions). If there is $|A| = 1$, then $|B|$ can be an arbitrary value which is not greater than the number of the neighboring nodes of A. If the value of $|B|$ is large, although the discovered place could meet Rule 3, the discovered hybrid model will be not realistic, since the number of splits (or joins) in a single block will be large. Therefore, we restrict the maximum value of $|A|$ (or $|B|$) to 3 in our implementations. This configuration will efficiently reduce the number of generated candidates (especially for complex DFGs). By exploiting this constraint, the number of generated candidates for the DFG in Figure 4 will be 15 rather than 21.

3.4 Data-level Constraints

We are trying to speed up our candidate pruning process by using light-weight filters at the trace and log levels, based on the data constraints implied in Rule 3.

Trace-level filter. Based on the execution semantics of Petri nets, we can deduce that if the activity sets A and B meet the requirement of Rule 3, it could also satisfy: $\forall \sigma \in T$, there is

$$\#_\sigma A = \#_\sigma B \qquad \text{(Rule 2)}$$

2. Note that block-structured process models can be seen as subset of workflow nets. However, we will be able to discover models that do not need to be block-structured, but still satisfy predefined requirements.

The frequencies of transitions in A match the frequencies of transitions in B, if at the end of each trace the place is empty. Similar to Rule 3, we use a parameter β to represent the approximation in Rule 2 and to check whether a candidate satisfies Rule 2 in most of the traces:

$$\beta = \frac{\#\sigma,\ \sigma : \textit{Rule 2}}{|T|} \qquad \text{(Approx. 2)}$$

For a given place candidate, computing the pruning process using Rule 2 will be much more lightweight than Rule 3, as Rule 2 does not require any prefix checking operations. We will use Rule 2 as a filter in HMD, i.e., if a candidate does not meet Rule 2, then the candidate will be pruned directly, so as to reduce the computation workloads for Rule 3. In the following, we refer this algorithm as $1f$-HMD.

Log-level filter. Let $\#_L A$ (or $\#_L B$) be the frequency of transitions in A (or B) appearing in the log L, similar to above, for a place candidate meeting Rule 2 and 3, the following condition could also happen:

$$\#_L A = \#_L B \qquad \text{(Rule 1)}$$

Namely, the overall sum of the frequencies of transitions in A matches the overall sum of the frequencies of transitions in B in a log level. Similarly, we use a parameter α defined as below to represent the approximation[3].

$$\alpha = \frac{|\#_L A - \#_L B|}{\#_L A + \#_L B} \qquad \text{(Approx. 1)}$$

We will use Rule 1 as a filter to reduce the computation workloads for Rule 2, therefore we can set a threshold t_1 here, and remove a candidate if $\alpha \geq t_1$. We call this algorithm as $2f$-HMD as it contains two filters.

3.5 Discussion

In terms of discovery performance in the presence of large event logs, using model-level constraints, our approach can efficiently reduce the number of place candidates in the step of candidate generation. In the meantime, using data-level constraints, we can use light-weight filters to pruning non-interesting candidates, i.e., if one of the two filters works, then the tested candidate will not go to the next phase. We need additional jobs to implement Rule 1 and Rule 2 in $2f$-HMD. However, as we will shown in our later section, most place candidates can be excluded quickly in our $2f$-HMD algorithm. It can obviously improve the performance of the HMD algorithm, without impacting the quality of discovered models, even in the cases that the values of t_1 are not theoretically *safe* (i.e., over-filtering could occur).

In fact, we can further speed up our model discovery by setting reasonable constraints on DFGs. For example, we can remove some arcs representing infrequent behavior in a log. These behavior occurs less frequent than

3. We can also use other mathematical expressions to describe the approximation, but this will not impact our approach in terms of implementations and bound analysis in the following section in principle.

"normal" behavior (e.g., the exceptional cases) and could be not interesting in some use cases. This kind of processing can simplify DFGs and consequently reduce the number of generated place candidates. Regardless, all related strategies can be treated as a pre-processing for our approach and a detailed discussion on such aspects will be outside the scope of this work. In our following implementation, we focus on discovering places over a DFG without removing any arcs from it.

4 Cloud-based Implementation

In this section, we present a detailed implementation of the proposed $2f$-HMD algorithm over Spark [8]. To cater to different DFGs in different process mining tools, we have separated the DFG discovery process from our HDM-based approaches and run it as an independent job. Namely, the implementations of the three algorithms are composed of two separate jobs. The source code we have used in this section and our evaluations is available at https://github.com/longcheng11/HybridM.

4.1 An overview of Spark

Spark [8] is a parallel computing platform very well suited to cloud computing. The reason is that it is elastic in terms of both storage (through the use of HDFS) and computation. This is in contrast with the conventional data systems which have to be carefully tuned to the specification of each node. In such scenarios, our implementation will be able to be easily deployed in a cloud to handle large event logs. Actually, the provided jar file at the link above is ready for cloud deployments.

We have picked Spark as the underlying framework rather than MapReduce [7], because Spark is becoming more and more popular, and it will largely replace MapReduce as the go-to model for big data analytics, as reported by Cloud Computing Technology Trends [9]. Another reason is that various data-parallel applications based on MapReduce can be expressed and executed efficiently using Spark. Here, we briefly introduce the fundamental data structure of Spark - the Resilient Distributed Datasets (RDD)

RDD is a central abstraction for Spark. It is a fault-tolerant and parallel data structure which lets users be able to store data in memory and control its partitioning [9]. Spark provides two types of parallel operations on RDDs [8]: *transformations* and *actions*. Transformations, including operations like *map* and *filter*, create a new RDD from the existing one. Actions, such as *reduce* and *collect*, conduct computations on an RDD and return the result to the driver program. Computation in Spark is expressed as functional transformations. Note that a RDD cannot be modified, however, a new RDD can be constructed by transforming an existing RDD.

4.2 Parallel Implementation

As described previously, the candidate generation and post-processing in our approach can be done on a single

Algorithm 2 DFG discovery in $2f$-HMD

The input log is read from underlying HDFS system, results in a RDD containing strings

1: val activities = log.**mapPartitions**(*parse trace*)
2: val pairs = activities.**mapPartitions**(iter ⇒ iter.**flatMap**(*generate pairs*))
3: val stat = pairs.**reduceByKey**(_ + _).**collect**()
4: Iteration over stat to extract directly-follows relations in the form of $(act1, iterable(act2, int))$
5: **Save** directly-follows relations on HDFS

node (i.e., master). Here, in terms of parallel implementations, we focus on the detailed implementations of DFG discovery and candidate pruning.

4.2.1 DFG Discovery

The discovery process can be implemented in a very simple way using Spark as shown in Algorithm 2. There, we also record the frequency for each directly-follows relationship. First, the activity log is placed in HDFS and read as a RDD. In this process, the log is automatically partitioned and each partition in the RDD is in the form of strings, where each string is a trace.

Following by that, we parse each trace into activities in the form of $iterable[act]$ by a map operation. This operation can be done in parallel at each partition of the RDD by using `mapPartitions` (line 1). The computation of the directly-follows relations in each trace can be done by scanning each $iterable[act]$ (i.e., parsed trace) and on that basis to generate pairs in the form of $\langle (act1, act2), 1\rangle$ for all neighboring activities. This can be also done in parallel as shown as line 2 in the algorithm. After that, we can aggregate all the pairs and calculate their global appearing frequency using aggregation functions. Here, we use the action `reduceByKey` (line 3). Namely, all the generated pairs on each partition will be first aggregated locally, and then redistributed over the computing systems based on their keys $(act1, act2)$, and finally aggregated again locally. This is more efficient than the action `groupBy` in Spark, especially when the log is large. The reason is that the first local aggregation can highly reduce the number of message pairs transferred over networks in the shuffling phase.

The aggregated results on each partition will be collected by the master node to construct the directly-follows graph. This can be done aggregating all the pairs in the form of $\langle act1, iterable(act2, int)\rangle$ by an iteration (line 4). This graph represents an abstraction of the whole log and will be used as an input to generate place candidates using the approach described in Section 3.2.2.

4.2.2 Candidate pruning

Parallelism. In the process of candidate pruning, as described in Section 3.2, we have to check all the place candidates over all the traces when applying Rule 2 and

Rule 3. The input log has been automatically partitioned when read from the HDFS system, therefore we will have two options on the parallel implementation. (1) Pipeline-based parallelism: all the candidates X are also (equal-size) partitioned, and each of the subsets are examined by each sublog (partition) in a pipeline way, and we can get the final results of the candidates in the subset when it passes the whole log. (2) Broadcast-based parallelism: which has implied in our Algorithm 1, i.e., all the candidates are broadcasted to all the sublogs and then the local checking results are merged to formulate the final outputs.

Theoretically, the first parallel approach will be more efficient than the second one, in terms of implementations using high performance computing (HPC) programming languages such as X10 [20], [21] and MPI [22]. This is because we can track the location of each subset of X and can manually assign them to a specified destination (i.e., sublog partition) according to our requirements. Moreover, we can let network communication and local computing work in a synchronous way, so as to extensively use the available system resources and consequently achieve a high performance. However, this is not straightforward for Spark (or a general cloud-based platform), since, unlike implementations using thread-level parallelism in HPC, Spark follows the data-flow model [23]. Namely, it does not allow fine-grained controls on data movement and computing.

A solution for the pipeline implementation is that we can first mark the candidates in each *partition* of the RDD of X with its partition index in Spark. For example, a candidate (A, B) in the partition 0 will be marked as $(0, (A, B))$. Then, we know that this candidate is in partition 0. In this case, we manually increase the index value by one after a candidate has been checked by a sublog and then shuffle all the candidates using *partitionBy* when all the local computing is done. This way each candidate can then go through the whole log. This approach would be efficient as well, regardless, the whole implementation needs to be done with multiple phases (i.e., the number of partitions of the input log), which could bring in obvious overheads in terms of communications and computation, due to the construction of new RDDs. Compared to this, the broadcast-based parallelism is easier to implement and will be more efficient. In fact, as we will show in our later evaluations, after the application of Rule 1, the number of generated candidates is much smaller than the underlying traces. Therefore, the parallelism problem can be mapped to a typical small-large case in cloud computing environments, and broadcasting has been shown to be very efficient way in such scenarios [24].

Algorithm 3 Candidate pruning in $2f$-HMD

All the place candidates X have been generated on the master node

1: val pairs = activities.**mapPartitions**(iter ⇒ iter.**flatMap**(*generate pairs1*))
2: val stat1 = pairs.**reduceByKey**(_ + _).**collectAsMap**()
3: Calculate the value α of each candidate using stat1
4: Prune the candidates in X if $\alpha > t_1$

5: val BX = sc.**broadcast**(X)
6: val stat2 = activities.**mapPartitions**(iter ⇒ {
7: **for** (trace ← iter) **yield** stat_map}).**mapPartitions**(
8: iter ⇒ { val LX = BX.value
9: 2D_Array = Check2(LX, iter) })
10: Collect stat2 and calculate β of each candidate
11: Prune the candidates in X if $\beta < t_2$

12: val BX = sc.**broadcast**(X)
13: Calculate stat3 based on replacing Check2 with Check3 by considering prefix checking in stat2
14: Collect stat3 and calculate γ of each candidate
15: Prune the candidates in X if $\gamma < t_3$

Implementation. Follow the above analysis, the detailed implementation of our candidate pruning is given in Algorithm 3. We first compute the frequency of each activity in the event log with two steps (lines 1-2): (1) generate pairs in the form of $\langle act, 1\rangle$ by going through all the activities using a map function, and (2) sum the number that an activity appears in all the sublog partitions by a reduce function. As the number of unique activities is very small and the number of candidates could be relatively large, we can collect the statistics information on the master node and then perform Rule 1. In this case, a candidate will go to the second step if the value of its α is not greater than the parameter t_1.

As shown in lines 5-11 in the algorithm, to implement the second filter, we first broadcast all the candidates to all the partitions (line 5). Then, in each partition we perform two operations: (1) create a map to record the activity frequency for each trace (line 7), and (2) check each candidate using Rule 2 (line 9). The checked results on each partition are recorded by pairs in the form of $\langle id, boolean\rangle$, where the id is the index of a candidate in X and the boolean records whether the candidate meets Rule 2 for a given trace. The index will be used to identify candidates after merging all the results from each partition. In this process, if none of the activities in A and B appears in a trace, then the checked results are not recorded. All the results will be collected finally to prune the candidates which do not meet Rule 2 (lines 10-11).

The parallel implementation of Rule 3 is very similar to Rule 2 (lines 12-15). The main difference is that each candidate on each partition has an additional prefix checking as described previously. The entire pruning process terminates when all local checked results are merged and compared with the threshold t_3. Based on the output places, the final hybrid process model can be then constructed as described in Section 3.2.

4.3 Safe Filtering Configuration

In our above implementation, we use a threshold t_2 in Rule 2 to pre-prune place candidates for Rule 3. As Rule 2 is implied by Rule 3, we should set t_2 to a lower value $t_2 \leq t_3$. If $t_2 \leq t_3$, then the filtering using Rule 2 does not influence the result, i.e., filtering is safe.

For the threshold t_1 in the Rule 1, obviously, the smaller the value t_1 is, the stronger the filter of Rule 1 will be. Therefore, to improve our performance, we need to set t_1 as small as possible. However, t_1 can not be set to a very small value (e.g., 0), otherwise Rule 1 will become a sufficient, but not necessary condition for Rule 2 and over-filtering will occur. To avoid this, we must guarantee that: for a given candidate (A, B), if $\beta \geq t_2$, then it should hold $\alpha \leq t_1$. Namely, to use Rule 1 as a filter in a safe way, we must set the threshold t_1 to value which is not smaller than the upper bound value of α.

Theorem 1: Rule 1 is a safe filter for Rule 2, if the threshold t_1 is set such that:

$$t_1 \geq \frac{(1-t_2)M}{2t_2 + (1-t_2)M} \tag{1}$$

where, M is the maximal length of traces in a log.

Proof: Let a candidate (A, B) with $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_m\}$. For the trace set T defined in Section 3.2.2, let the i-th trace T_i contain $|T_i|$ activities, and the number of appearances of the j-th transitions in A (and B) be a_{ij} (and b_{ij}). Rule 1 is a necessary condition for Rule 2, according to Rule 2, there are $K = |T|\beta$ traces meeting the condition $\#_\sigma A = \#_\sigma B$ and $|T|(1-\beta)$ traces do not. Based on this, for the simplicity of our mathematical presentation, we sort all the traces, and keep that the first K traces satisfy $\#_\sigma A = \#_\sigma B$ while the rest do not. Then, we have:

$$\#_L A = \sum_{i=1}^{K}\sum_{j=1}^{n} a_{ij} + \sum_{i=K+1}^{|T|}\sum_{j=1}^{n} a_{ij} \tag{2a}$$

$$\#_L B = \sum_{i=1}^{K}\sum_{j=1}^{m} b_{ij} + \sum_{i=K+1}^{|T|}\sum_{j=1}^{m} b_{ij} \tag{2b}$$

$$\sum_{j=1}^{n} a_{ij} = \sum_{j=1}^{m} b_{ij}, \quad \forall i \in [1, K] \tag{2c}$$

$$\sum_{j=1}^{n} a_{ij} + \sum_{j=1}^{m} b_{ij} \leq |T_i| \tag{2d}$$

Based on the definition of α and Eq. (2a-2c), there is:

$$\alpha = \frac{|\sum_{i=K+1}^{|T|}(\sum_{j=1}^{n} a_{ij} - \sum_{j=1}^{m} b_{ij})|}{2\sum_{i=1}^{K}\sum_{j=1}^{n} a_{ij} + \sum_{i=K+1}^{|T|}(\sum_{j=1}^{n} a_{ij} + \sum_{j=1}^{m} b_{ij})} \tag{3}$$

Obviously,

$$\alpha \leq \frac{\sum_{i=K+1}^{|T|}\sum_{j=1}^{n} a_{ij}}{2\sum_{i=1}^{K}\sum_{j=1}^{n} a_{ij} + \sum_{i=K+1}^{|T|}\sum_{j=1}^{n} a_{ij}} = \alpha_1 \tag{4}$$

We assume that the average frequency of activities in A in a trace T_i ($i \in [K+1, |T|]$) is y, from Eq. (4) we have:

$$\alpha_1 = \frac{(|T|-K)y}{2\sum_{i=1}^{K}\sum_{j=1}^{n} a_{ij} + (|T|-K)y} \tag{5}$$

Moreover, assume that for the first K traces, the average appearing time of the activities in A (also B) in a trace T_i ($i \in [1, K]$) is x. Based on Eq. (5), there is:

$$\alpha_1 = \frac{(|T|-K)y}{2Kx + (|T|-K)y} = \frac{(1-\beta)y}{2\beta x + (1-\beta)y} \tag{6}$$

According to our definitions, we have $x \geq 1$, otherwise there exists at least one trace T_i in T that $A \cap T_i = \emptyset \wedge B \cap T_i = \emptyset$. Moreover, we have $\beta \geq t_2$, therefore, we have:

$$\alpha_1 \leq \frac{(1-t_2)y}{2t_2 + (1-t_2)y} \tag{7}$$

From Eq. 2(d), we have $\forall i, y \leq Max\{|T_i|\}$. As T is a subset of L, there is $Max\{|T_i|\} \leq Max\{|L_i|\} = M$. Therefore, based on Eq. (4) and (7), we have Eq. (1). ■

Remark: Based on Eq. (1), we can say that for a log with the longest trace containing 10 activities, if we set $t_2 = 0.9$, then we should set t_1 to 0.36 to guarantee that Rule 1 can always safely filter the place candidates, regardless of what the candidates and the underlying log look like. Moreover, we can see that the larger M is, the larger the value of t_1 should be, i.e., the weaker the filter will be. In term of pruning performance, it should be noticed that Eq. (1) just gives the theoretically worst condition. As we will show in our evaluation in Section 5, setting t_1 to a value which is smaller than the theoretical value, we can still get the same places, with less time.

5 Evaluation

In this section, we present an experimental evaluation of our approaches.

5.1 Experimental Framework

We have implemented the HMD, $1f$-HMD and $2f$-HMD algorithms using Scala over Spark [8].

Platform. We evaluate our approach over a cluster. Each node we used has 4 CPU cores running at 2.80 GHz with 16GB of RAM. The operating system is Linux kernel version 2.6.32-279 and the software stack consists of Spark version 2.0.0, Hadoop version 2.7.3, Scala version 2.11.4 and Java version 1.7.0_25. As the computational infrastructure uses non-virtualised computational resources, we believe that our testbed approximates commercial offerings in terms of cloud computing [24], and thus can characterize how our algorithms will behave in a cloud. In particular, our test enviroment closely approximates the Bare Metal servers in IBM Softlayer [25] and the Cluster instances in Amazon EC2 [26].

Datasets. We run our performance tests over different datasets based on real life logs taken from the Business

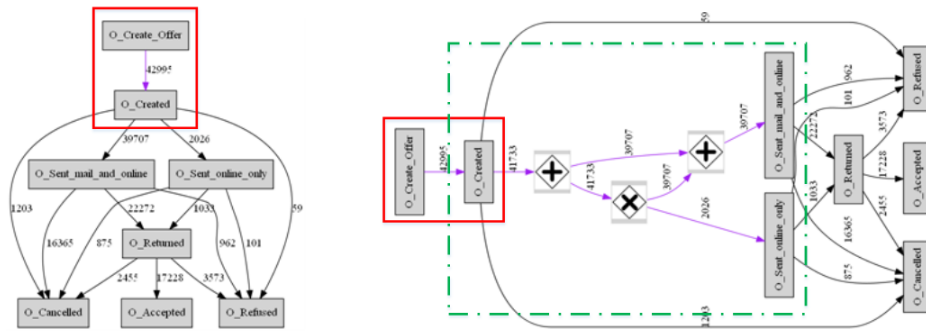


Fig. 5: The two discovered hybrid process models for Log 2 with two different input sets of parameters.

Process Intelligence Challenge (BPIC) of the year 2017. There are two event logs: (1) the *application* event log [18], which contains 31,509 traces, the number of events is 1,202,267 and the maximum number of events in a trace is 180; and (2) the *offer* event log [27], which contains 42,995 traces, the number of events is 193,849 and the maximum number of events in a trace is 5. These two logs represent a large and small as well as a complex and simple case (in terms of DFG complexity), for hybrid process model discovery. For simplicity, we refer the two event logs as Log 1 and Log 2 respectively.

Setup. We set the following system parameters for Spark: *spark_worker_memory* and *spark_executor_memory* are set to 15GB and *spark_worker_cores* is to 4. There are three parameters t_1, t_2 and t_3 in our implementation, i.e., the thresholds for Rule 1, 2 and 3 respectively. Consider the safety and performance of the two filters in our f-HMD algorithms, as a default we set t_2 to a value equal to t_3, and t_1 to the value as we described in our theoretical analysis in Section 4.3. We will vary the value of t_3 and use 90% as the default.

In all our experiments, we read input files from the HDFS system [17] and output discovered hybrid process models (in `dot` file format) to the disk of the master node. We measure runtime as the elapsed time from job submission to the job being reported as finished. As a default, we implemented our tests using 9 nodes, including one master node and 8 worker nodes (i.e., 32 cores).

5.2 Experimental Results

In this subsection, we first present the discovered hybrid models using our algorithms. Then, we report their runtime performance and scalability.

5.2.1 Discovered Models

We have evaluated our approach over Log 1 and Log 2 with different input parameters. The discovered models in dot (and PDF) format are included in the distribution of our software. For simplicity, we only present the discovered models for Log 2 here.

With two different parameter configurations $(0, 1)$ and $(0.217, 0.9)$ for (t_1, t_2), the two discovered hybrid models using the $2f$-HMD algorithms for Log 2 are depicted in Figure 5. It can be seen that our approach can indeed discover hybrid process models. These discovered models highlight the unique feature of hybrid models: they provide a mix of formal (highlighted by rectangle blocks) and informal constructs. In both the hybrid models, there are precise semantics between the activities `O_Create_Offer` and `O_Create`. Namely, the latter activity always follows the former one in underlying business processes of Log 2. Additionally, we can see that the discovered model with input parameter $(0, 1)$ is much simpler than the one with $(0.217, 0.9)$, i.e., the left model has not formal gateways whereas the right model has three gateways. The reason is that the selectivity of places is very high when t_2 equal to 1: the discovered precise parts (models) should follow the Petri net semantic in a strict way (i.e., without approximation), and any possible abnormal behavior or noise in a trace or any uncompleted traces (because the log is captured in a certain period of time) will results in a potential candidate being pruned by the rules. In comparison, the approximation used in the configuration with $(0.217, 0.9)$ shows more tolerances in such conditions and thus more places are discrovered. It should be noted that such kinds of tolerance could make a discovered model unsound [1] (e.g., the green/dashed block in the model), regardless, we can refine this either manually or using advanced techniques based on our requirements, and detailed analysis in such aspect will be out the scope of this work.

5.2.2 Efficiency

Runtime. We conducted our tests using different input parameters (t_1, t_2) and Figure 6 shows the detailed runtime of each algorithm with different input parameters over the two logs. It can be seen that: (1) The DFG discovery, i.e., Job 1, can be done quickly and it takes only seconds for both the logs. (2) Using filters, for Log 1, 1*f*-HMD algorithm is faster than HMD, and 2*f*-HMD always performs the best for different input parameters. Regardless, for Log 2, all the algorithms perform nearly the same. As we will explain later, the reason is that the workload of Log 2 is comparably small for underlying systems. Furthermore, for Log 1, the performance advantage of our 2*f*-HMD algorithms becomes more obvious

(t_1, t_2)	(0.957, 0.8)			(0.909, 0.9)			(0, 1)		
Alg.	**Job1**	**Job2**	**Rt.**	**Job1**	**Job2**	**Rt.**	**Job1**	**Job2**	**Rt.**
2f	5	192	**197**	6	156	**162**	5	6	**11**
1f	5	270	275	6	270	276	6	270	276
HMD	6	294	300	6	294	300	5	288	293

(a) runtime over Log 1 (sec)

(0.385, 0.8)			(0.217, 0.9)			(0, 1)		
Job1	**Job2**	**Rt.**	**Job1**	**Job2**	**Rt.**	**Job1**	**Job2**	**Rt.**
5	7	12	5	9	14	5	9	14
5	8	13	5	8	13	5	8	13
5	8	13	5	7	12	5	7	12

(b) runtime over Log 2 (sec)

Fig. 6: Runtime of each algorithm over different logs with different input parameters in safe conditions (32 cores)

(t_1, t_2)	(0.957, 0.8)			(0.909, 0.9)			(0, 1)		
Rule	***2f***	***1f***	**HMD**	***2f***	***1f***	**HMD**	***2f***	***1f***	**HMD**
0	5967	5967	5967	5967	5967	5967	5967	5967	5967
1	5173	/	/	4819	/	/	5	/	/
2	39	39	/	25	25	/	5	5	/
3	20	20	20	13	13	13	4	4	4

(a) number of candidates for Log 1

(0.385, 0.8)			(0.217, 0.9)			(0, 1)		
2f	***1f***	**HMD**	***2f***	***1f***	**HMD**	***2f***	***1f***	**HMD**
57	57	57	57	57	57	57	57	57
28	/	/	17	/	/	1	/	/
7	7	/	3	3	/	1	1	/
7	7	7	3	3	3	1	1	1

(b) number of candidates for Log 2

Fig. 7: Detailed number of candidates for each algorithm with different input sets of parameters (32 cores)

(t_1, t_2)	(0.05, 0.9)	(0.1, 0.9)	(0.2, 0.9)	(0.5, 0.9)	(0.909, 0.9)
0	5967	5967	5967	5967	5967
Rule 1	293	508	896	1936	4819
Rule 2	22	25	25	25	25
Rule 3	11	13	13	13	13
time (sec)	16	17	20	41	162

(a) detailed executions for Log 1

(0.01, 0.9)	(0.05, 0.9)	(0.1, 0.9)	(0.2, 0.9)	(0.217, 0.9)
57	57	57	57	57
1	7	10	15	17
1	3	3	3	3
1	3	3	3	3
14	14	12	13	14

(b) detailed executions for Log 2

Fig. 8: Number of candidates and the runtime of each execution for $2f$-HMD in unsafe conditions (32 cores)

with increasing values for t_2. These can be explained by the fact that the job of DFG discovery is simple. In the meantime, the computing workload can be efficiently reduced using the two filters, and the filtering power becomes stronger with the increase of t_2. Specifically, when t_2 is set to it possible maximum value (i.e., 1), the job 2 of the $2f$-HMD algorithm can be done in seconds and much faster than other two approaches. In contrast, we can see that the runtime *1f*-HMD is nearly constant with varying the values of input parameters, the possible reason is that the number of input candidates is huge and the computing cost over Rule 2 dominates the runtime of the approach.

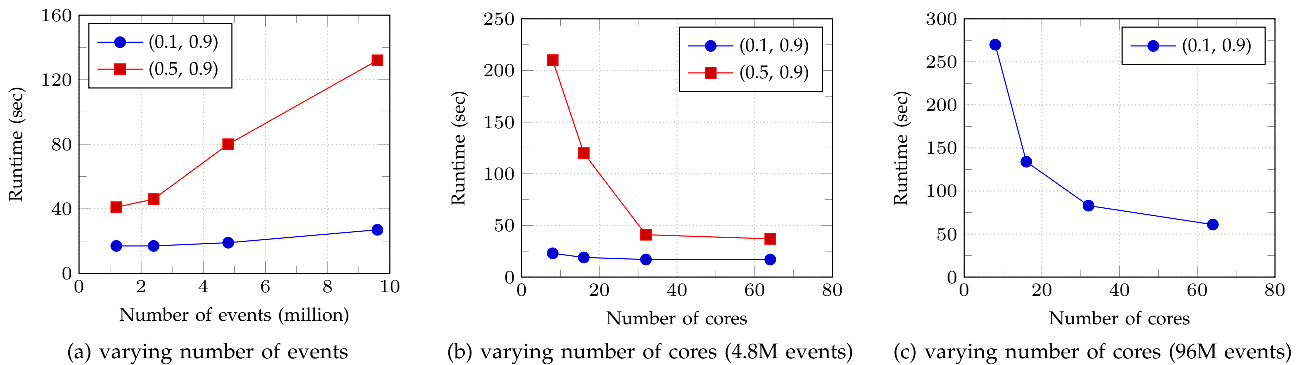
Power of filters. We recorded the detailed number of candidates in each phase of all the three algorithms in our executions, and the results are given in Figure 7. There, for Log 1, the results demonstrate that large number of place candidates (i.e., workloads) can be pruned using filters for both the logs. For example, with parameters 0.957 and 0.8, the number can be reduced to 39 using two filters, compared to the 5967 in HMD. Moreover, when increasing the values of the parameters, the numbers can be further reduced (e.g., to 5 using *2f*-HMD for final case), and this brings in great performance improvement for the Log 1 as we have observed in Figure 6(a). Similar to Log 1, we can see that the number of candidates is also reduced using filters for Log 2. Regardless, this improvement is not obvious, as the original number of input candidates is small (i.e., 57). That is also the reason why the runtime is nearly the same for the three algorithms with different input parameters as shown in Figure 6(b).

Safe vs. unsafe. To avoid over-filtering, we have set t_1 to a safe value (i.e., the worst case) in above tests, according to our previous theoretical analysis. To examine the impact of t_1, in terms of correctness of outputs and performance of the $2f$-HMD algorithm, we vary its value in some unsafe conditions and report the results in Figure 8. There, we fix the value of t_2 to 0.9, and it can be seen that in a wide range (e.g., 0.1 vs. 0.909 for Log 1) of t_1, the number of discovered places is the same (e.g., 13 after Rule 3 for Log 1) for each case. Not only is the number of places the same, the set of places are the same. This implies that aggressive filtering has no effect on the outcome although this approach is theoretically unsafe. In the meantime, with decreasing the value of t_1, we can also observe that the runtime is decreasing obviously for Log 1. The reason is that the power of the first filter is becoming stronger, which can be observed on the changes of the number of candidates in Figure 8(a). In comparison, the runtime over Log 2 does not change, due to the original workload is small as we have analyzed. Additionally, we see that over-filtering could indeed happen if t_1 is set to a very small value (e.g., 0.05 for Log 1).

5.2.3 Scalability

We tested the scalability of the $2f$-HMD algorithm by varying both the size of input logs and the number of execution cores. Since we are targeting large and complex logs, we only report the results over the datasets from Log 1. For generality, we fix the value of t_2 to 0.9 and select the cases with t_1 equals to 0.1 and 0.5 respectively (both are proved to be *safe* on the basis of our above results).

Number of events. To evaluate our approach for larger logs, we first fix the number of workers to 8 (32 cores)



(a) varying number of events (b) varying number of cores (4.8M events) (c) varying number of cores (96M events)

Fig. 9: The scalability of 2f-HMD by varying the size of input logs and the number of execution cores with different input sets of parameters over datasets from Log 1.

and increase the size of Log 1 by duplicating its traces, from 31,509 (1.2 million events) to 252,072 (9.6 million events). As shown in Figure 9(a), we can see that the runtime is increasing with increasing the number of events for both the configurations, and the trend for the case with $t_1 = 0.5$ is more obviously than $t_1 = 0.1$. Moreover, both cases achieve superlinear performance. The reason is that numbers of candidates are highly reduced in both cases, which makes the workload relatively small for the underlying systems though the number of events is increased.

Number of cores. As shown in Figure 9(b), we fix the number of events to 4.8 millions and double the number of cores from 8 cores to 64 cores (16 nodes) for the conditions with t_1 equals to 0.1 and 0.5. It can be seen the runtime of 2f-HMD algorithm decreases with increasing the number of cores in both the cases. Regardless, the runtime does not behave very well for the case with $t_1 = 0.1$, and also for the case $t_2 = 0.5$ when increasing the number of cores from 32 to 64. We believe that the reason is the computing workload is comparably small for the underlying platform. To prove this, we increase the workloads for the case with $t_1 = 0.1$ by increasing the number of traces to 2.5 millions (with 96 million events) and run the tests again. From Figure 9(c), it can be seen that the runtime scales very well, although the workload is still relatively small when the number of cores reaches 64. Specifically, it can be seen that 2f-HMD can discover the hybrid model from about 100 million events, in about 1 minute using 64 cores (16 nodes).

6 Related Work

Process discovery is one of the most challenging process mining tasks. State-of-the-art techniques can already deal with logs where each process instance is recorded as a case with ordered events and that each event is related to exactly one case by a case identifier. Examples of algorithms that learn process models based on event data include α-miner [19], Heuristic Miner [28], ILP Miner [29] and Inductive Miner [5]. Most of these methods describe in scientific literature focus on discovering process models having formal semantics. This is in stark constraint with most of the commercial tools which discover informal models, not having clear semantics. The reason is that Petri nets (and the like) are perceived as complex and also force people to be very explicit about the ordering of activities. However, when there is infrequent behavior and splits and joins are clear cut, it does not make any sense to straitjacket reality in a very precise model. In fact, none of the 20+ commercial tools is explicit about the type of splits and joins. In comparison to this, we introduce an efficient way to discovery hybrid models by adding formal semantics to the informal models, which can be easily discovered and widely used. The approach presented here is related to the approach in [16] which also uses hybrid process models (hybrid Petri nets). In this paper we focus on distributing the discovery of such process models in a cloud computing environment. Moreover, there are also differences in the discovery approach and the notation used.

With the growing of big data, large-scale event logs start to challenge current process mining techniques [30]. The reason is that sampling technique could lead to statistically valid results on mainstream behavior [31], but would not lead to insights into the exceptional behavior, which is typically the goal of process mining [32]. Some commercial tools, such as Celonis exploit modern database techniques to handle big event log. However, as the amount of event log data continues to grow, such an approach will be no longer feasible and it will be impossible to process the entire data set on a single machine, due to the hardware limitations (e.g., CPU and memory). To handle this issue, we need to resort to distributed and cloud-based computing platforms (see Chapter 12 in [1]). In fact, such platforms (or frameworks) such as MapReduce and Spark have been used to support event correlation discovery, i.e., to identify the events that are part of the same case [33], [34]. Moreover, in the context of process discovery, MapReduce has also been used to implement the α-miner and the Flexible Heuristics Miner [35], [36]. Although all these approaches have shown that they can achieve obviously speedups in the presence of big event log, different from them, we focus on discovering hybrid process models in this work.

We have implemented our approach on Spark (over HDFS) and experimentally shown that the proposed $2f$-HMD algorithm is efficient and scalable in a cloud computing enviroment. We believe that the evaluation conducted in this work and the described results are of value to the community as a basis for understanding the merits of the approach. Since our computing is conducted on partitioned logs, our approach can be easily extended in more complex computing environments, such as that hybrid model discovery in multi-cloud (or cross-orgnization) environments [37]. In such a case, to efficiently use the available computing resources, we can further divide the filter operations of $2f$-HMD into seperate jobs and apply advanced scheduling strategies on them (e.g., [38]) to achieve the best possible performance.

7 Conclusions

In this paper, we have introduced an efficient approach, called f-HMD, to discover hybrid process models from large event logs. We have described the detailed implementation of our approach in a cloud computing enviroment (i.e., over Spark) and our experimental results have demonstrated that the proposed $2f$-HMD algorithm is very efficient and scalable.

Our future work lies in extending our method with more advanced techniques to get better hybrid models (e.g., avoid unsound models to avoid the situation depicted in Figure 5). Moreover, we will employ parameter tuning strategies in our implementation to select safe filtering parameters for each individual candidate in a dynamial way, and consequently to enhance the robustness and efficiency of our prototype in production environments. Finally, we plan to incorporate the proposed approach in our prior event correlation system [34], to develop a high-performance process discovery system which can automatically discover hybrid process models from homogeneous (and heterogeneous) event logs in large-scale distributed cloud scenarios.

Acknowledgments

This work was supported by the NWO DeLiBiDa research program. Long Cheng thanks the support of the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 799066. Wil van der Aalst thanks the Alexander von Humboldt (AvH) Stiftung for supporting his research.

References

[1] W. van der Aalst, *Process Mining: Data Science in Action*. Springer, 2016.

[2] W. van der Aalst and E. Damiani, "Processes meet big data: Connecting data science with process science," *IEEE Trans. Serv. Comput.*, vol. 8, no. 6, pp. 810–819, 2015.

[3] S. Goedertier, J. De Weerdt, D. Martens, J. Vanthienen, and B. Baesens, "Process discovery in event logs: An application in the Telecom industry," *Applied Soft Computing*, vol. 11, no. 2, pp. 1697–1710, 2011.

[4] S. Suriadi, M. T. Wynn, C. Ouyang, A. H. ter Hofstede, and N. J. van Dijk, "Understanding process behaviours in a large insurance company in Australia: A case study," in *CAiSE*, 2013, pp. 449–464.

[5] S. J. Leemans, D. Fahland, and W. M. van der Aalst, "Discovering block-structured process models from event logs-a constructive approach," in *International Conference on Applications and Theory of Petri Nets and Concurrency*, 2013, pp. 311–329.

[6] W. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science-Research and Development*, vol. 23, no. 2, pp. 99–113, 2009.

[7] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Networked Systems Design and Implementation*, 2012, pp. 15–28.

[9] L. Cheng and S. Kotoulas, "Efficient skew handling for outer joins in a cloud computing environment," *IEEE Trans. Cloud Comput.*, vol. 6, no. 2, pp. 558–571, 2018.

[10] Disco, https://fluxicon.com/disco/.

[11] Celonis, http://www.celonis.com/en/.

[12] ProcessGold, http://www.processgold.com/en/.

[13] Minit, https://www.minit.io/.

[14] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[15] N. Lohmann, E. Verbeek, and R. Dijkman, "Petri net transformations for business processes–a survey," in *Transactions on Petri Nets and other Models of Concurrency II*, 2009, pp. 46–63.

[16] W. van der Aalst, R. De Masellis, C. D. Francescomarino, and C. Ghidini, "Learning hybrid process models from events: Process discovery without faking confidence," in *Proc. Int. Conf. Business Process Management*, 2017.

[17] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010, pp. 1–10.

[18] B. van Dongen, "BPI Challenge 2017." [Online]. Available: doi:10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b

[19] W. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 9, pp. 1128–1142, 2004.

[20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[21] L. Cheng, A. Malik, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Fast compression of large semantic web data using X10," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2603–2617, 2016.

[22] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[23] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 443–454.

[24] L. Cheng, I. Tachmazidis, S. Kotoulas, and G. Antoniou, "Design and evaluation of small–large outer joins in cloud computing environments," *Journal of Parallel and Distributed Computing*, vol. 110, pp. 2–15, 2017.

[25] softlayer, http://www.softlayer.com/.

[26] Amazon, https://aws.amazon.com/ec2/.

[27] B. van Dongen, "BPI Challenge 2017 - Offer log." [Online]. Available: doi:10.4121/uuid:7e326e7e-8b93-4701-8860-71213edf0fbe

[28] A. Weijters, W. van der Aalst, and A. A. De Medeiros, "Process mining with the heuristics miner algorithm," *Technische Universiteit Eindhoven, Tech. Rep. WP*, vol. 166, pp. 1–34, 2006.

[29] J. M. E. van der Werf, B. F. van Dongen, C. A. Hurkens, and A. Serebrenik, "Process discovery using integer linear programming," in *Petri Nets*, 2008, pp. 368–387.

[30] S. Sakr, Z. Maamar, A. Awad, B. Benatallah, and W. M. Van Der Aalst, "Business process analytics and big data systems: A roadmap to bridge the gap," *IEEE Access*, vol. 6, pp. 77 308–77 320, 2018.

[31] C. Liu, Y. Pei, Q. Zeng, and H. Duan, "LogRank: An approach to sample business process event log for efficient discovery," in *Proc. 11th Int. Conf. Knowledge Science, Engineering and Management*, 2018, pp. 415–425.

[32] A. Syamsiyah, B. F. van Dongen, and W. van der Aalst, "DB-XES: enabling process discovery in the large," in *SIMPDA*, 2016, pp. 63–77.

[33] H. Reguieg, B. Benatallah, H. R. M. Nezhad, and F. Toumani, "Event correlation analytics: Scaling process mining using MapReduce-aware event correlation discovery techniques," *IEEE Trans. Serv. Comput.*, vol. 8, no. 6, pp. 847–860, 2015.

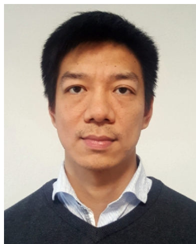
[34] L. Cheng, B. van Dongen, and W. van der Aalst., "Efficient event correlation over distributed systems," in *Proc. 17th IEEE/ACM Int. Symp. Cluster, Cloud and Grid Comput.*, 2017, pp. 1–10.

[35] S. Hernández, J. Ezpeleta, S. J. v. Zelst, and W. van der Aalst, "Assessing process discovery scalability in data intensive environments," in *Proc. IEEE/ACM Int. Symp. Big Data Comput.*, 2015, pp. 99–104.

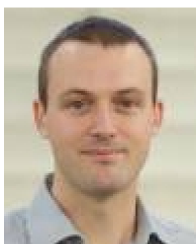
[36] J. Evermann, "Scalable process discovery using Map-Reduce," *IEEE Trans. Serv. Comput.*, vol. 9, no. 3, pp. 469–481, 2016.

[37] C. Liu, H. Duan, Z. Qingtian, M. Zhou, F. Lu, and J. Cheng, "Towards comprehensive support for privacy preservation cross-organization business process mining," *IEEE Trans. Serv. Comput.*, no. 1, pp. 1–1, 2016.

[38] B. Lin, W. Guo, N. Xiong, G. Chen, A. V. Vasilakos, and H. Zhang, "A pretreatment workflow scheduling approach for big data applications in multicloud environments," *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 3, pp. 581–594, 2016.



Long Cheng is a Marie Curie Fellow at the Performance Engineering Laboratory in University College Dublin (UCD). He received a B.E. from Harbin Institute of Technology, China in 2007, a M.Sc from University of Duisburg-Essen, Germany in 2010 and a Ph.D from National University of Ireland Maynooth in 2014. He has worked at organizations such as Huawei Technologies, IBM Research, TU Dresden and TU Eindhoven (TU/e). His research interests mainly include parallel and distributed computing, big data, Semantic Web and process mining.



Boudewijn van Dongen is a full Professor in computer science and chair of the Process Analytics group at Eindhoven University of Technology (TU/e). The research group distinguishes itself in the Information Systems discipline by its fundamental focus on modeling, understanding, analyzing, and improving processes. Boudewijn's research focuses on conformance checking: anything where observed behavior needs to be related to previously modeled behavior.

Wil van der Aalst a full Professor at RWTH Aachen University leading the Process and Data Science (PADS) group. He is also part-time affiliated with the Eindhoven University of Technology (TU/e). Until December 2017, he was the scientific director of the Data Science Center Eindhoven (DSC/e) and led the Architecture of Information Systems group at TU/e. Since 2003, he holds a part-time position at Queensland University of Technology (QUT). Currently, he is also a visiting researcher at Fondazione Bruno Kessler (FBK) in Trento and a member of the Board of Governors of Tilburg University. His research interests include process mining, Petri nets, business process management, workflow management, process modeling, and process analysis. He is a senior member of the IEEE.