Fodina: a Robust and Flexible Heuristic Process Discovery Technique

Seppe K.L.M. vanden Broucke[a,*], Jochen De Weerdt[a]

[a]*Research Center for Management Informatics (LIRIS), KU Leuven*
Naamsestraat 69, B-3000, Leuven, Belgium

Abstract

In this paper, we present Fodina, a process discovery technique with a strong focus on robustness and flexibility. To do so, we improve upon and extend an existing process discovery algorithm, namely Heuristics Miner. We have identified several drawbacks which impact the reliability of existing heuristic-based process discovery techniques and therefore propose a new algorithm which is shown to be better performing in terms of process model quality, adds the ability to mine duplicate tasks, and allows for flexible configuration options.

Keywords: process mining, process discovery, event logs

1. Introduction

The digital revolution that is taking place is significantly changing the way industry and people manage, store and analyze the vast amounts of data that is being generated and processed. Naturally, the challenge in this "big data" environment is to be able to extract value and insights from these

*Corresponding author
Email addresses: seppe.vandenbroucke@kuleuven.be (Seppe K.L.M. vanden Broucke), jochen.deweerdt@kuleuven.be (Jochen De Weerdt)

information repositories in an effective manner. In the context of business process management, where processes are responsible for the correct undertaking of system functionalities, end users hence desire to extract process-oriented aspects that can enable a better understanding of the reality observed. The field of Process Mining aims to offer solutions to tackle this core task: starting from so called "event logs", containing footprints or "traces" of real process executions, process mining techniques aim at discovering, analyzing and enhancing process models (van der Aalst, 2011).

From its arising, the process mining field has evolved into several directions, with *process discovery* perhaps being the most challenging task, as demonstrated by the large amount of techniques available nowadays. What makes process discovery difficult is the fact that derived process models should perform well over four quality dimensions: fitness (ability of the model to reproduce the traces in the event log), precision (how precise is the model in representing the behavior in the log), generalization (is the model able to generalize for behavior not in the log) and simplicity (the well-known Occam's Razor principle). Doing so is difficult as event logs can contain noise and erroneous behavior, in which case a *robust* discovery algorithm should be able to deal with such behavior. Additionally, users oftentimes want to impose criteria regarding the layout or quality focus of discovered models (e.g. in terms of precision version generalization), so that *flexibility* of configuration is a desireable but hard-to-achieve trait in process discovery as well. In this work, we present *Fodina*, a process discovery technique with a strong focus on robustness and flexibility. The primary contribution of this paper is not to propose another process discovery technique, but rather to pragmatically improve upon a class of existing process discovery algorithms, namely the so-called "heuristic" miners (Weijters et al.,

2006), adding some particular interesting features to make the approach *more robust to noisy data*, add the ability to *discover duplicate activities*, and allow for *flexible configuration* options to drive the discovery according to end user input. Heuristics Miner is one of the best known and most used process discovery algorithms both by practitioners and researchers, and has also proven its worth in benchmarking studies illustrating the technique's ability to discover high-quality models (De Weerdt et al., 2012). However, we have identified various problematic issues, which negatively impact the reliability of the technique. As such, we perform a thorough review of the existing Heuristics Miner with all its variants to identify a list of issues and consequently propose a new implementation of a heuristic process miner which retains the ability to discover high-quality models in a fast manner, whilst being more robust to noise, can discover duplicate activities, and contains configuration options to drive the discovery according to end user input.

The remainder of this paper is structured as follows. Section 2 provides an overview of related work in the literature and introduces preliminary concepts. Section 3 lists the identified issues present in existing works. Next, we introduce Fodina. Section 5 compares the new implementation with other techniques based on an experimental evaluation. Section 6 concludes the paper.

2. Preliminaries

2.1. Literature Overview and Related Work

In the area of process discovery, the α-*algorithm* can be regarded as one of the most fundamental techniques; Van der Aalst et al. prove that the tech-

nique is able to learn an important class of workflow nets (structured workflow nets) from event logs (van der Aalst et al., 2004), provided that the given event log is sufficiently complete and that the event log does not contain any noise. In order to deal with the problem of noise of the α-algorithm class of techniques, Weijters et al. developed *Heuristics Miner* (Weijters et al., 2006). This technique extends the α-algorithm in that it applies frequency information with regard to relationships between activities in an event log and is able to mine a wider set of process model constructs. Invisible activities (task present in the model but not in the event log), however, are not mined directly as such, but the mined "Heuristics net" does not specifically require the presence of invisible activities to model activity skips or complex routing constructs (after conversion to a Petri net, the model will then contain the invisible activities necessary to represent these constructs). Duplicate activities (tasks in the model logged under the same event label) are also not mined. Note that other process discovery techniques also make use of Heuristics nets to represent mined process models, most notably Genetic Miner (Alves de Medeiros et al., 2007), although this technique is not regarded as a typical heuristic process discovery algorithm as it applies an evolutionary optimization strategy to derive a fitting process model, rather than applying frequency-based dependency measures. In 2010, Burattin and Sperduti proposed an adaptation of the Heuristics Miner algorithm, *Heuristics Miner++*, which extends the former by considering activities with time intervals, i.e. having a starting and ending time instead of being logged as an atomic, zero-duration event (Burattin and Sperduti, 2010). The same authors have also proposed a modified Heuristics Miner which is able to deal with streaming event data (Burattin et al., 2012).

Weijters and Ribeiro have also created a modified version of their Heuris-

tics Miner algorithm, *Flexible Heuristics Miner* (Weijters and Ribeiro, 2011), which outputs the mined model as a Causal net (van der Aalst, 2011). Although this representation is very similar to Heuristics nets, an important difference exists in the way input and output bindings are expressed for each task.

2.2. Definitions

Process discovery starts from a so-called event log and outputs a process model using a particular representational language. In order to obtain a usable event log, it is assumed that it is possible to record events so that each event refers to an activity (e.g. "sign order"), a process instance (e.g. "PI101") and that the events are ordered, either based on an absolute time stamp or on the basis of relative ordering (a sequence number). In some cases, the specific state transition of the activity is also recorded in the event, for example to denote when an activity was started versus its time of completion. As mentioned above, some process discovery algorithms, like Heuristics Miner++, do take into account the duration of an activity (e.g. to determine if an activity's execution overlaps with another one), but most discovery algorithms only consider a process instance as a sequence of (atomic) events. As such, we will make use of the following notation.

Definition 1. Event Logs—Let event log L be defined as a multiset of traces (process instances). The cardinality (or size) of an event log $|L|$ denotes the total number of traces in the log. $|\bigcup L|$ represents the size of the set over the event log, i.e. the number of unique traces, not counting duplicates. A trace $\sigma \in L$ is a finite sequence of events with length $|\sigma|$ and with σ_i the event at position i in trace σ. The number of times a trace σ appears in L is called the multiplicity of the trace. Since an ordering is explicitly defined

between events in a sequence and the related process instance can be left implicit, the events themselves can simply be denoted based on their activity name, e.g. $\sigma = \langle a, b, c, d \rangle$. The set of activities occurring in the event log is then denoted as $T_L = \{\sigma_i | \sigma \in L, i = 1 \ldots |\sigma|\}$ (the activity alphabet of the event log).

Process models mined by heuristic dependency-based process discovery techniques are frequently expressed in the form of a Causal net (C-net).

Definition 2. Causal Nets—A Causal net is a tuple $C_N = (T_C, t_s, t_e, I, O)$ where T_C is a finite set of tasks modeled by the Causal net, $I : T_C \mapsto \{X \subseteq \mathcal{P}(T_C) | X = \{\emptyset\} \vee \emptyset \notin X\}$ defines the set of possible input bindings per task (an input binding is a set of sets of activities) and $O : T_C \mapsto \{X \subseteq \mathcal{P}(T_C) | X = \{\emptyset\} \vee \emptyset \notin X\}$ defines the set of possible output bindings per task. Causal nets must have a start task $t_s \in T_C$ for which $I(t_s) = \{\emptyset\}$ and one end task $t_e \in T_C$ for which $O(t_e) = \{\emptyset\}$. For each task $t \in T_C$, $\Box t = \bigcup(I(t))$ takes the union of all subsets in $I(t)$ and denotes the set of all input tasks, whereas $t\Box = \bigcup(O(t))$ denotes the set of output tasks of t. Based on this, a dependency graph (T_C, D) can be defined as a relation on T_C, with D the set of pairs: $\{(a, b) | a \in T_C \wedge b \in T_C \wedge (a \in \Box b \vee b \in a\Box)\}$. All tasks $t \in T_C$ in the graph (T_C, D) should lie on a path from the starting to the ending task. The set of sets of tasks denoting the input and output bindings (I and O respectively) are interpreted as a disjunction (between the sets of activities) of conjunctions (between the activities within a set). The output bindings for each task create obligations whereas input bindings resolve obligations. A "binding sequence" models an execution path through a Causal net starting and ending with the start and end task respectively and while removing all obligations created during execution. As an example, consider a task t with

$I(t) = \{\{a\}, \{b\}\}$ and $O(t) = \{\{c, d\}, \{e\}\}$, meaning that this activity can only be executed when it is preceded by a *or* b (disjunction between sets), and that its execution creates the obligation that the task should be followed with c *and* d (conjunction within sets), *or* just e.

Further details on the semantics of Causal nets can be found in (van der Aalst et al., 2011). Three important remarks should still be mentioned, however. First, note that the semantics of Causal nets are non-local, as an output binding may create the obligation to execute an activity much later in the process. In addition, in the case of an output binding consisting of multiple sets of sets of tasks, it is not clear at the time of executing the task at hand which of the possible conjunctive AND sets will be resolved later on. As such, during execution, a state must be kept represented by multi-sets of pending obligations which still need to be resolved. Second, note that some descriptions and implementations of heuristic, dependency-based process discovery algorithms, such as Heuristics Miner (Weijters et al., 2006), derive models in a similar representational language, i.e. a Heuristics net, but impose an inverted interpretation on the input and output bindings, namely as a conjunction of disjunctions, meaning that $O(t) = \{\{c, d\}, \{e\}\}$ then denotes that t must be followed by e *and* either c *or* d. This representation introduces some issues which will be discussed in more detail below. Finally, we remark that although Causal nets represent a well-defined and formal representational language for process models, they are nevertheless converted to Petri nets (another formal representational language for concurrent models) in most practical applications (van der Aalst et al., 2011). The main reason behind this being that most process mining techniques and implementations offer more mature support for Petri net analysis than for Causal nets.

2.3. Heuristic Dependency-based Process Discovery

This section outlines the (core) workings of existing heuristic dependency-based process discovery algorithms such as Heuristics Miner (Weijters et al., 2006) and Flexible Heuristics Miner (Weijters and Ribeiro, 2011). Put broadly, the steps of these discovery algorithms all execute the following four steps. First, counts of "basic relations" between activities in the event log are derived. Following information can trivially be abstracted from the event log (assume a and b are activities $\in T_L$): $|a|$: the number of times activity a appears in the event log (the frequency of a); $|a > b|$: the direct succession count between a and b (the number of times that a is directly followed by b); $|a >> b|$: the repetition count between a and b (the number of times that a is directly followed by b and b again followed by a); $|a >>> b|$: the indirect succession count between a and b (the number of times that a is eventually followed by b, *but before the next appearance of* a *or* b). Note that every direct succession is also counted towards the indirect succession count.

Next, a dependency graph is constructed using "dependency measures" or "causal metrics", describing the basic causal semantics between activities (follows and precedes relations). Based on user-defined thresholds, a dependency (an arc between two activities) is added in the dependency graph between two activities when a dependency measure exceeds this threshold. Various suitable measures have been proposed in the literature, either in the context of a heuristic dependency-based process discovery algorithm (Weijters et al., 2006; Weijters and Ribeiro, 2011), or in related work where the concept of activity dependencies is also utilized, e.g. in (Maruster et al., 2006), where such metrics are used as the inputs to construct a data set to be used in a rule learning task.

Third, the semantic information, i.e. the sets of input and output bind-

ings per activity (representing the XOR and AND splits and joins) is mined. In the original definition of the Heuristics Miner algorithm (Weijters et al., 2006), a separate measure is applied to derive confidence towards two tasks occurring in parallel. In the Flexible Heuristics Miner algorithm, XOR and AND relations are mined in a different manner (Weijters and Ribeiro, 2011). Simply put, for an activity a with depending activities b and c, counts are calculated corresponding with the number of times a was followed by b only, c only or by both b and c. Based on the frequency of the different possible "patterns" (i.e. $\{\{b\},\{c\}\}$ and $\{\{b,c\}\}$), an output binding is chosen. The exact procedure on how this final decision is made is left unspecified in (Weijters and Ribeiro, 2011); available implementations of the discovery algorithm include all discovered split and join patterns in the final causal net, making the approach less robust to noise.

In a final, optional step, the long-distance dependencies are mined. To do so, another dependency metric and threshold are defined, using the value of $|a >>> b|$. However, many activity-pairs exist for which this metric will return a high value (e.g. between the starting activity and many other activities), although no additional dependency should be added. Therefore, a check is typically performed to see whether it is possible to go from task a to the ending task in the dependency graph without having visited b. If this is possible then the additional long-distance dependency is added to the dependency graph (rendering it more precise).

3. Identified Issues

The value of the existing (Flexible) Heuristics Miner algorithm should not be understated, due to its robustness to noise, ease-of-interpretation

and speed. As of now, it remains one of the most often applied and best performing process discovery algorithms (De Weerdt et al., 2012). However, several issues can still be identified which open up opportunities for solid improvements. Some of these issues are due to a vague or incomplete definition, whereas other lie in an incorrect implementation. We order the issues based on whether they relate to the discovery of the model, the semantics of the Heuristics nets model itself, or due to other implementation aspects. We base our discussion on the implementation found in the latest versions of ProM 6.6.

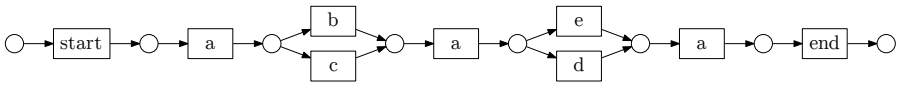
3.1. Model Discovery: Unconnected Tasks

(Flexible) Heuristics Miner includes an "all tasks connected" heuristic, ensuring that each task in the dependency graph has at least one incoming and outgoing arc (except for the start and end tasks). To do so, the best candidate task (i.e. using the highest $|a > b|$ value) is taken to determine the primal causal and dependent task. However, even when using this approach, the particular complexity of several event logs (such as the "hospital log" used in the BPI 2011 Challenge[1]) causes some tasks to remain unconnected with the rest of the model.

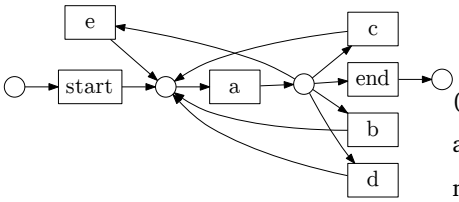
3.2. Model Discovery: Duplicate Tasks

No current heuristic process discovery algorithm is able to mine duplicate activities. The ability to detect duplicate tasks could nevertheless greatly improve the understandability and structural clarity of the obtained process model. To illustrate why this is the case, Figure 1 shows a comparison between an original Petri net model and the (correctly converted) Petri

[1]DOI: doi:10.4121/d9769f3d-0ab0-4fb8-803b-0d1120ffcf54.



(a) Original Petri net model.



(b) More complex Petri net model after converting mined Heuristic net.

Figure 1: An original Petri net model and result after mining on generated event log and converting to Petri net with Heuristics Miner.

net model after running Heuristics Miner. The fact that all activities sharing the same label are treated as a single task in the process model leads to the creation of extra arcs and dependencies and thus more structurally complex results. Even although the mined Petri net does perfectly fit the behavior in the event log, the ability to discover duplicate tasks would greatly improve the understandability and clarity of discovered process models.

3.3. Model Discovery: Long-distance Dependencies

The way long-distance dependencies are constructed differs somewhat in the actual implementation of Heuristics Miner compared to the approach as described in the literature (Weijters and Ribeiro, 2011). In the implementation, the $|a >>> b|$ count between two tasks is only incremented once within the same trace, although multiple occurrences of the same $a >>> b$ pattern can exist within the same trace. Furthermore, the current long-distance dependency definition in Heuristics Miner is somewhat overly sen-

sitive regarding the actual setting of the threshold to be applied. That is, lowering the threshold allows to discover more long-distance dependencies, but also causes redundant long-distance dependencies to show up in the resulting process model, for instance between the starting task and another task, making the resulting net harder to interpret.

3.4. Model Discovery: Split and Join Semantics

The way splits and joins are mined in the currently available set of heuristic discovery algorithms also warrants some attention. Although the recommended method to mine the AND and XOR relations in the input and output bindings is to make use of pattern-based techniques as described in (Weijters and Ribeiro, 2011), one implementation of the Heuristics Miner ("Mine for a Heuristics Net using Heuristics Miner" in ProM 6.6) uses the non-flexible metric-based technique as indicated in Section 2.3. A second implementation ("Mine for a Causal Net using Heuristics Miner" which is hidden in the UI in ProM 6.6) does use pattern-based frequency counting to discover and annotate the split and join semantics, but includes *each* seen pattern in the resulting Causal net, which makes the implementation sensitive to noise in this regard.

3.5. Model Semantics: ICS Fitness Calculation

The manner by which fitness is reported for mined Heuristics nets is not particularly well described; the fitness reported in the implemented Heuristics Miner is often referred to in literature as the Improved Continuous Semantics (ICS) fitness. To be exact, the fitness measure reported is the $PF_{complete}$ measure as described by (Alves de Medeiros, 2006), with traces being parsed using a continuous semantics token game, meaning that

the execution of a trace is continued after the occurrence of a non-fitting activity. The exact manner however how this parsing (or replay) of traces occurs is not clearly described in literature. In addition, there is another implementation-related aspect which warrants mentioning in this context: after calculating the ICS fitness, a final operation is performed ("disconnectUnusedElements()"), which changes the structure of the net and can also influence the fitness of the Heuristics net. The ICS fitness is, however, not recalculated to reflect these changes.

3.6. Implementation: Incorrect Conversion to Petri Nets

Heuristic nets are frequently converted to Petri nets for additional analysis, such as e.g. determining the level of conformance between an event log and the model, as most process mining techniques offer more mature support for Petri net analysis than for other model representations. The current implementation of Heuristics Miner contains a faulty implementation of a Heuristics net to Petri net convertor ("Convert Heuristics net into Petri net", ProM 6.6). The reason behind this issue is an erroneous interpretation of the semantics of the input and output bindings for Heuristics nets, which differ from those found in Causal nets; within the Heuristics net, the input and output bindings of an activity represent a conjunctive set of disjunctive subsets. In addition, there is another intricacy present regarding the semantics of Heuristic nets which is oftentimes forgotten: selecting an activity in one subset also implies the selection of the same activity if it appears in other subsets. E.g. an activity a with output bindings $\{\{b, e\}, \{c, e\}\}$ should be interpreted as "(one-of b XOR e) AND (one-of c,e)", but disallows choosing combinations where e only appears in one of the subsets and where "e AND e" is reduced to "just e".

4. Robust and Flexible Heuristic Process Discovery with Fodina

Based on a thorough literature study, containing a breakdown of all available and currently applied heuristic process discovery variants (Section 2) and inspecting the issues present in these variants (Section 3) as outlined above, we propose a new heuristic process discovery algorithm, named Fodina, which aims to provide a robust iteration of this set of techniques in order to mine Causal nets, including also some new features which will be discussed in the remainder of this section.

4.1. Process Discovery with Fodina

An overview of the steps performed by Fodina to mine a Causal net is given as follows:

1. Convert the event log to a "task log". Contextual information is used to (optionally) mine duplicates;
2. Derive counts of "basic relations" between activities in the event log;
3. Construct a basic dependency graph using dependency measures;
4. Set the start and end task in the dependency graph;
5. Resolve binary conflicts in the dependency graph (optional);
6. Assure each task is reachable in the dependency graph (optional);
7. Mine long-distance dependencies in the dependency graph (optional);
8. Mine the semantic information, i.e. the sets of input and output bindings per activity to convert the dependency graph to a Causal net.

4.1.1. Steps 1 and 2: Construct Task Log and Derive Basic Relations

In the first step, the given event log is converted to a "task log", where each activity in the event log (i.e. in T_L) is mapped to a task to-be included

in the resulting Causal net (i.e. to T_C). Without mining duplicate tasks, this mapping is trivial ($T_C = T_L$). When the option is set to mine duplicates, the same activity in the event log can be mapped to multiple tasks in T_C. To determine which activities should be duplicated, we apply a strategy inspired by Genetic Miner (Alves de Medeiros, 2006). In this technique, it is assumed that duplicate tasks can be distinguished based on their local context, meaning the set of input and output elements of the duplicates. The aim of Genetic Miner is then to mine process models in which duplicates of a same task do not have input or output elements in common. This approach has a couple of benefits. For example, parsing Causal nets with duplicate tasks remains relatively simple, because the context (the prefix and postfix in the trace) of an event is sufficient to choose which duplicate task to fire. Similar approaches have also been applied before, e.g. in (Lu et al., 2016). Based on this, we have included a procedure which directly infers the duplicate tasks from the given event log, applying the same principle of a local context. Say that we are trying to derive if an activity a in the event log L should be duplicated. We construct a set of contexts $C = \{(\sigma_{i-1}, \sigma_i, \sigma_{i+1}) | \sigma \in L, \sigma_i = a\}$. Next, we construct the set of grouped contexts $\mathcal{C} = \{C' \in \mathcal{P}(C) | \forall (x, y, z) \in C' : \nexists (i, j, k) \in C \backslash C' : j = y \wedge (i = x \vee k = z)$, which corresponds with the duplicate tasks to be placed in the Causal net. As an example, consider the event log: $\{\langle start, a, b, c, a, d, e, a, end\rangle, \langle start, a, c, b, a, d, e, a, end\rangle, \langle start, a, b, c, a, e, d, a, end\rangle, \langle start, a, c, b, a, e, d, a, end\rangle\}$. The set of contexts for activity a is then equal to $\{(start, a, b), (start, a, c), (c, a, d), (b, a, d), (c, a, e), (b, a, e), (e, a, end), (d, a, end)\}$. The set of grouped contexts is constructed so that the local contexts of a are separated so that they do not overlap with one another: $\{\{(start, a, b), (start, a, c)\}, \{(c, a, d), (b, a, d), (c, a, e), (b, a, e)\}, \{(e, a, end), (d, a, end)\}\}$, representing three duplicate tasks.

Note that deriving duplicate tasks like this might indeed lead to the duplication of activities which could nevertheless be kept as a single task in the process model without impacting fitness or heavily affecting understandability. However, the choice is made to ignore this, as these "redundant" duplicate tasks still remain easy to interpret in the final process model.

To mitigate against noise leading towards the derivation of undesired duplicate tasks (consider for example an activity which is inserted at a random position in the event log and is thus likely to be surrounded by an unseen context), we introduce a "duplicate task threshold", which works as follows: when a duplicate task is created with a frequency which is below the duplicate task threshold ratio t_{dup} (the frequency of this particular duplicate task over the frequency of all duplicate tasks), the duplicate task for this particular context is removed and merged with the duplicate task having the greatest frequency. Note that an alternative strategy consists of ignoring such noisy events altogether, though we consider such "cleaning" of event logs (i.e. removing infrequent activities in the traces altogether) as a pre-discovery task which should be executed before invoking Fodina (or any other discovery algorithm).

In addition, an option was added to better allow the duplication of activities which also repeat. Consider for example again the trace $\langle start, a, a, a, b, a, a, a, end\rangle$. Based on this, the following set of grouped contexts would be constructed for a: $\mathcal{C} = \{\{(start, a, a), (a, a, a), (b, a, a), (a, a, end)\}\}$, i.e. the context (a, a, a) causes that no duplicate tasks can be found for activity a. Therefore, we allow to "collapse" repeated tasks during the derivation of duplicates, so that the two duplicate tasks for a can then be discovered (before b and after b, i.e. based on the collapsed trace $\langle start, a, -, -, b, a, -, -, end\rangle$, we derive $\mathcal{C} = \{\{(start, a, b)\}, \{(b, a, end)\}\}$).

For the remainder of this paper, we assume a mapper function $\mu : T_L \mapsto T_C$ which is able to unambiguously map activities occurring in traces in an event log to a task occurring in the causal net. Given the way duplicate tasks are dealt with, μ is able to map an activity to one single task in the causal net by inspecting the local context of this activity.

The second step (derivation of basic relation counts) is performed completely similar as done in Heuristics Miner (see Subsection 2.3), making sure, however, to correctly derive the $|a >>> b|$ information, i.e. by counting multiple occurrences of an $a >>> b$ pattern in the same trace.

4.1.2. Steps 3 to 6: Construction of the Dependency Graph

Using T_C, μ and basic relation counts, a dependency graph can be constructed. Algorithm 1 provides a formal overview of these steps. In lines 1-8, arcs are introduced for length one loops, normal dependencies and length two loops respectively. Note that for normal dependencies, we do apply a different measure compared to Heuristics Miner (line 3), as we argue that the direct succession of a task b after a is not always suitable direct counter-evidence against the direct succession of b after a. (The metric now also lies in the range $[0, 1]$.) For length one and length two loops, we retain the measures of Heuristics Miner.

All associated thresholds in Fodina (t_{l1l}, t_d and t_{l2l}) operate separately from each other during the construction of the dependency graph, which is not the case in the Heuristics Miner implementation, where changing one threshold might have no effect without also lowering other thresholds, which in turn might cause other undesired dependencies to show up. We have also removed the "positive observations" and "relative-to-best" thresholds in Fodina, as it was observed that their impact is negligible in most

Algorithm 1 Steps 3 to 7 of Fodina: construction of the dependency graph (continued on next page).

Input: An event log L with activity set T_L, a set of tasks T_C with mapping $\mu : \sigma \in L \mapsto T_C$, basic relations $|a > b|$, $|a >> b|$, and $|a >>> b|$, settings t_d, t_{l1l}, t_{l2l}, t_{ld} (thresholds), $noL2lWithL1l$, $noBinaryConflicts$, $connectNet$, and $mineLongDependencies$.

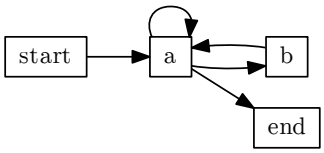
Output: A dependency graph D with start and end tasks t_s and t_e; a set $ldeps$ indicating which dependencies are long distance.

1: $D \leftarrow \{\}$
2: $\forall a \in T_C : \frac{|a>a|}{|a>a|+1} \geq t_{l1l}, D \leftarrow D \bigcup \{(a, a)\}$
3: $\forall a, b \in T_C : \frac{|a>b|}{|a>b|+|b>a|+1} \geq t_d, D \leftarrow D \bigcup \{(a, b)\}$
4: $\forall a, b \in T_C : \frac{|a>>b|+|b>>a|}{|a>>b|+|b>>a|+1} \geq t_{l2l} \wedge (\neg noL2lWithL1l \vee (a, a) \notin D \vee (b, b) \notin D)$, $D \leftarrow D \bigcup \{(a, b), (b, a)\}$
5:
6: $t_s \leftarrow \text{argmax}_{x \in T_C} \sum_{\sigma \in L : x = \mu(\sigma_1)} 1$
7: $t_e \leftarrow \text{argmax}_{x \in T_C} \sum_{\sigma \in L : x = \mu(\sigma_{|\sigma|})} 1$
8: $\forall a \in T_C, D \leftarrow D \setminus \{(a, t_s), (t_e, a)\}$
9:
10: **if** $noBinaryConflicts$ **then**
11: **for** $a, b \in T_C : (a, b) \in D \wedge (b, a) \in D$ **do**
12: $D \leftarrow D \setminus \{(a, b), (b, a)\}$
13: **if** $|a >> b| > 0$ **then** $D \leftarrow D \bigcup \{(a, a)\}$
14: **if** $|b >> a| > 0$ **then** $D \leftarrow D \bigcup \{(b, b)\}$
15: **for** $c \in T_C : c \neq a \wedge c \neq b$ **do**
16: **if** $(c, a) \in D \vee (c, b) \in D$ **then** $D \leftarrow D \bigcup \{(c, a), (c, b)\}$
17: **if** $(a, c) \in D \vee (b, c) \in D$ **then** $D \leftarrow D \bigcup \{(a, c), (b, c)\}$

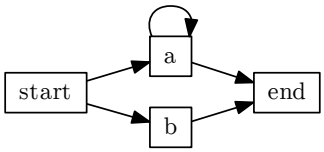
cases (or covered by the other thresholds). During the discovery of length two loops to add to the dependency graph, users have the option to prohibit a length two loop dependency between a and b (i.e. from a to b and b to a) when these two tasks are both already involved in a length one loop with themselves ($noL2LWithL1L$ in line 4). This can be beneficial in cases

18: **if** $connectNet$ **then**
19: **repeat**
20: % US, UE is the set of unconnected activities from the start/end task respectively
21: **if** $US \neq \emptyset$ **then**
22: $(bestin, un) \leftarrow \text{argmax}_{(c,u):c \in T_C, u \in US \wedge (c,u) \notin D \wedge c \neq t_e \wedge u \neq t_s} \frac{|c>u|}{(|c>u|+|u>c|+1)}$
23: $D \leftarrow D \bigcup \{(bestin, un)\}$
24: **if** $UE \neq \emptyset$ **then**
25: $(un, bestout) \leftarrow \text{argmax}_{(u,c):c \in T_C, u \in UE \wedge (u,c) \notin D \wedge u \neq t_e \wedge c \neq t_s} \frac{|u>c|}{(|c>u|+|u>c|+1)}$
26: $D \leftarrow D \bigcup \{(un, bestout)\}$
27: **until** All tasks lie on path from start to end
28:
29: $ldeps \leftarrow \{\}$
30: **if** $mineLongDependencies$ **then**
31: **for** $a, b \in T_C : \frac{2|a>>b|}{|a|+|b|+1} - \frac{2||a|-|b||}{|a|+|b|+1} \geq t_{ld}$ **do**
32: **if** $PathExistsFromToWithoutVisiting(t_s, t_e, a) \wedge$
33: $PathExistsFromToWithoutVisiting(t_s, t_e, b) \wedge$
34: $PathExistsFromToWithoutVisiting(a, t_e, b)$ **then**
35: $D \leftarrow D \bigcup \{(a, b)\}$
36: $ldeps \leftarrow ldeps \bigcup \{(a, b)\}$

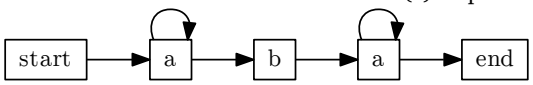
where both activities are length one loops and both are depending in an AND relation on the same, third activity, leading to observations such as $\langle start, a, b, a, a, a, b, b, a, b, b, b, end \rangle$. This trace can then be configured to be modeled in two ways, either with a length two loop (and a XOR split/join for $start$ and end) or without a length two loop (with an AND split/join being inferred for $start$ and end in step 8). In the fourth step, the start and end tasks are set in the dependency net (based on start/end frequency in the traces of the log; all incoming and outgoing arcs of start and end activities respectively are removed from the dependency graph, lines 6-8). If desired, users can first pre-process an event log L to add artificial starting and end-



(a) Dependency net obtained without mining for duplicate tasks and allowing for "binary conflicts". All splits and joins are in a XOR relation.



(b) Dependency net obtained without mining for duplicate tasks and with the "resolve binary conflict" option enabled, converting the length two loop to an AND split/join.



(c) Dependency net obtained with mining for duplicate tasks enabled.

Figure 2: Different dependency graph outcomes obtained with the Fodina miner under various configurations for the trace $\langle start, a, a, a, b, a, a, a, end \rangle$.

ing activities, which are prepended and appended respectively to each trace in the event log.

As stated above, during the discovery of length two loops, users may prohibit a length two loop dependency when the two associated tasks are both already involved in a length one loop with themselves. Additionally, applying the concept of "binary conflicts" as described in (Günther, 2009), users have the option to enable Fodina to try to convert *all* length two loops to a single AND relation whenever possible (lines 10-17). As such, the trace $\langle start, a, a, a, b, a, a, a, end \rangle$, for example, can now be mined in three different ways, as depicted by Figure 2, all of which fit the given trace.

The following, optional, step assures that each task in the dependency graph is connected, as is a requirement for a valid Causal net (lines 18-27). This is not implemented by checking if each task has at least one input and output arc in the dependency graph, similar as done by Heuristics Miner,

but rather by continuing to add the next-best dependency graph edge (i.e. the edge with the highest dependency measure connecting the unconnected task with another task) until all tasks lie on a path between the start and end activity. This approach is more time consuming than the simple check performed by Heuristics Miner, but prevents the discovery of nets containing disconnected elements.

The final step of mining long-distance dependencies is also optional (lines 29-36), and is performed here *before* the mining of the semantic AND and XOR relations. We use the same dependency measure as the one described for Flexible Heuristics Miner (Weijters and Ribeiro, 2011), i.e. $\left(\frac{2\times(|a>>b|)}{|a|+|b|+1}\right) - \left(\frac{2\times abs(|a|-|b|)}{|a|+|b|+1}\right)$. To better avoid the mining of unnecessary long-distance dependencies, we not only perform a check to see whether it is possible to go from a to the end task without visiting b (if b is always visited, the long-distance dependency is unnecessary), but also evaluate whether it is possible to go from the start to end task without visiting a or without visiting b (similarly, if a or b is always visited, the long-distance dependency is unnecessary, see lines 32-34). Only if all these checks pass, the candidate long-distance dependency is introduced in the Causal net.

4.1.3. Step 8: Mine Split and Join Semantics

Finally, the dependency graph is converted to a Causal net by mining the AND and XOR relations to construct the input and output bindings. Algorithm 2 describes our approach, which is comparable to the pattern-based approach of Flexible Heuristics Miner, but adds configurable options to make the discovery more robust to noise.

To construct the output binding for a task, for example, we count the number of times each pattern (i.e. a possible, particular subset of out-

Algorithm 2 Step 8 of Fodina: mining split and join semantics (continued on next page).

Input: An event log L with activity set T_L, a dependency graph D with a set of tasks T_C with mapping $\mu : \sigma \in L \mapsto T_C$ and with t_s, t_e start and end tasks; a set $ldeps$ indicating which dependencies are long distance. Threshold t_{pat} between -1 and 1.

Output: Input and output bindings I_t, O_t for each task $t \in T_C$.

for $t \in T_C$ **do**
  $I_t \leftarrow$ FINDPATTERNS(t, $input$)
  $O_t \leftarrow$ FINDPATTERNS(t, $output$)

function FINDPATTERNS(t, dir)
  $PS \leftarrow \{\}$ % Set of extracted patterns
  if $dir =$ input **then** $C \leftarrow \{x \in T_C | (x,t) \in D\}$ % Set of connected tasks
  else if $dir =$ output **then** $C \leftarrow \{x \in T_C | (t,x) \in D\}$
  for $\sigma_i \in \sigma, \sigma \in L : \mu(\sigma_i) = t$ **do**
    % Task found: construct input/output pattern at this position
    $P \leftarrow \{\}$
    for $c \in C$ **do**
      if $dir =$ input **then**
        $CO \leftarrow \{x \in T_C | (c,x) \in D\}$ % Set of output tasks for candidate input task
        $cp \leftarrow \max\{j \in [i-1 \ldots 0] | \mu(\sigma_j) = c\}$
        if $cp \wedge \nexists k \in [cp+1 \ldots i-1] : \mu(\sigma_k) = t \vee (\mu(\sigma_k) \in CO \wedge (c,t) \notin ldeps)$ **then**
          $P \leftarrow P \bigcup \{c\}$
      else if $dir =$ output **then**
        $CI \leftarrow \{x \in T_C | (x,c) \in D\}$ % Set of input tasks for candidate output task
        $cp \leftarrow \min\{j \in [i+1 \ldots |\sigma|] | \mu(\sigma_j) = c\}$
        if $cp \wedge \nexists k \in [cp-1 \ldots i+1] : \mu(\sigma_k) = t \vee (\mu(\sigma_k) \in CI \wedge (t,c) \notin ldeps)$ **then**
          $P \leftarrow P \bigcup \{c\}$
    $PS \leftarrow PS \bigcup \{P\}$ % Add the constructed pattern to PS
    Increment pattern count $|P|$ by one or set to zero if first time seen
  return FILTERPATTERNS(t, PS, C)

26: **function** FILTERPATTERNS(t, PS, C)
27: $PF \leftarrow \{\}$ % Set of retained patterns
28: **if** $|PS| > 0$ **then**
29: $tr \leftarrow \sum_{P \in PS} \frac{|P|}{|t| \times |PS|}$ % $|P|$ indicates the number of times this pattern was seen, $|t|$ indicates the number of times this activity occurs in the log, $|PS|$ indicates the total number of patterns found
30: **if** $t_{pat} \leq 0$ **then** $tr \leftarrow tr + t_{pat} * tr$
31: **else**$tr \leftarrow tr + t_{pat} * (1 - tr)$
32: **for** $P \in PS : \frac{|P|}{|t|} \geq tr$ **do**
33: $PF \leftarrow PF \bigcup \{P\}$
34: **for** $c \in C : \nexists P \in PS : c \in P$ **do**
35: $PF \leftarrow PF \bigcup \{\{c\}\}$
36: **return** PF

put tasks) was found after the occurrence of this task, but (i) only up until the next occurrence of the task under consideration and (ii) where the task under consideration was also the nearest input task for every task in the pattern (lines 6-26). For instance, consider a simple dependency graph $D = \{(s,a),(a,a),(a,b),(a,c),(b,c).(b,e),(c,e)\}$ and a single trace $\sigma = \langle s,a,a,b,c,a,c,b,e\rangle$. We now wish to count the number of times each pattern occurred for the outputs of a. We hence loop over every occurrence of a in the trace and inspect its output pattern up until the next occurrence of a. For the first occurrence, we hence check $\langle s, \underline{a \succ, a}, b, c, a, c, b, e\rangle$. The only output task occurring up until the next occurrence of a is a itself, and the current a is also the closest input task for that a, so we increase the count with one for pattern $\{a\}$. For the next occurrence, we check $\langle s, a, \underline{a \succ, b, c, a}, c, b, e\rangle$. Here, b, c and a all occur as outputs. For b and a, the currently inspected a is the closest input task, but for c, b lies closer, so that the resulting output pattern is $\{a,b\}$. Finally, we check

$\langle s, a, a, b, c, \underline{a \succ, c, b, e}\rangle$; b and c occur as output tasks here, both of them now having a as their nearest (working backwards from their position) input task, so that the final pattern is $\{b, c\}$. The output bindings in the Causal net are hence $O(a) = \{\{a\}, \{a, b\}, \{b, c\}\}$ based on this single trace.

If one of the output tasks is a long-distance dependent task, however, the nearest input criterion is skipped for this task (lines 17 and 22), as the task under consideration can never be the nearest input (by definition). This edge-case of including long-distance dependencies in the calculation of split and joins is not included in the description of the Flexible Heuristics Miner. Another improvement relates to the way patterns are selected for inclusion in the Causal net. First, every pattern with a frequency ratio exceeding a configurable threshold is selected (instead of all found patterns, lines 27-34). Next, the remaining output tasks in the dependency graph which are *not* included in any output binding (in the thus-far selected patterns) are added as singleton subsets to the output binding (lines 35-37). Increasing the thresholds thus leads to the selection of less patterns (only the frequent patterns are selected), with potentially more output activities remaining which are then added as singleton subsets. Fodina has been implemented as a ProM 6 plugin and is available with source code at `http://www.processmining.be/fodina`.

4.2. Heuristic Execution Semantics for Causal Nets

Next to the process discovery task, the process mining research field describes a second important analysis task, denoted as conformance checking, where existing process models are compared with behavior as captured in event logs so as to measure how well a process model performs with respect to the actual executions of the process at hand. As such, the “goodness” of

a process model is typically assessed over the quality dimensions of fitness (or: recall, sensitivity), precision (or: appropriateness), generalization, and simplicity (or: structure, complexity).

In order to determine the quality of process models mined with Fodina in accordance with the given event log (or a new log), we define an execution semantic for Causal nets, similar to the semantics used by the Improved Continuous Semantics (ICS) measure in Heuristics Miner (Alves de Medeiros, 2006). That is, we implement a heuristic, *greedy* replay procedure as follows. This procedure is less permitting (i.e. not non-local) than the theoretical case of finding a possible binding sequence for a trace (van der Aalst et al., 2011), though much faster and, due to the nature of heuristic discovery algorithms, not limiting in practice. During the replay, a state is kept, representing a set of *pending obligations* which must be fulfilled by future tasks. Each obligation is expressed as a tuple of the form (t, o), i.e. the obligation to resolve the set of output bindings o that followed after the execution of t, so that state $S = \{(t, o)|t \in T_C, o \subseteq O(t)\}$. As such, S is initially empty when replaying a trace σ. Next, all events $\sigma_i \in \sigma$ are iterated and fired. Each time an event is fired, the matching task among the duplicates in the Causal net is chosen (i.e. $\mu(\sigma_i)$). Then, for the selected task which is to be fired, the best input binding $i \in I(\mu(\sigma_i))$ is determined based on the amount of unsatisfied (i.e. "missing") input tasks being present. An input task x for an input binding is unsatisfied when $\nexists(t, o) \in S|t = x \wedge \mu(\sigma_i) \in \bigcup(o)$ with $\bigcup(o)$[2]. Naturally, the most optimal input binding is one which has no missing input tasks and can thus fire with-

[2]The union operator here defines the flattening of a set of sets, e.g. $\bigcup(\{\{a, b\}, \{a, c\}\}) = \{a, b, c\}$.

out error; in the case where multiple input bindings can be satisfied, the one containing the largest amount of tasks is selected. If there are missing input tasks, we continue but indicate the execution of this activity as being "force fired", i.e. with errors.

After firing, a successor state is generated as follows. First of all, the fired task $\mu(\sigma_i)$ is removed from all pending obligations which contain the fired task in one of their output bindings: $\forall(t, o) \in S, b \in o | \mu(\sigma_i) \in \bigcup(o)$, update binding $b := b \setminus \mu(\sigma_i)$ if $\mu(\sigma_i) \in b, \emptyset$ elsewise. Note that output bindings which do not contain the fired task are emptied altogether, as they represent a part of the disjunction of obligations that cannot be resolved anymore (this emphasizes the greedy nature of the replay). Before moving on to the next event in the trace, the state is updated with a new obligation containing the output bindings of the event which was fired, i.e. $S = S \cup \{(\mu(\sigma_i), O(\mu(\sigma_i))\}$. At the end of trace replay, it is possible that the final state contains leftover, pending obligations, either due to the local nature of the replay algorithm or due to the discovered Causal net not being sound. It is up to the replay measure used whether to punish on this aspect.

We emphasize that the replay procedure described here is *greedy and hence heuristic*. Nevertheless, for Causal nets mined with Fodina (and other heuristic miners), this replay semantic is able to correctly parse the traces contained in the event log. As a simple example, consider the dependency graph in Figure 2(b). The trace $\langle start, a, a, a, b, a, a, a, end \rangle$ is now replayed as follows (the chosen best input binding is indicated in bold face):

Task σ_i	$I(\sigma_i)$	$O(\sigma_i)$	S after firing σ_i
$start$	$\{\{\emptyset\}\}$	$\{\{a, b\}\}$	$\{(start, \{\{a, b\}\})\}$
a	$\{\{\mathbf{start}\}, \{a\}\}$	$\{\{a\}, \{end\}\}$	$\{(start, \{\{b\}\}), (a, \{\{a\}, \{end\}\})\}$
a	$\{\{start\}, \{\mathbf{a}\}\}$	$\{\{a\}, \{end\}\}$	$\{(start, \{\{b\}\}), (a, \{\{a\}, \{end\}\})\}$
a	$\{\{start\}, \{\mathbf{a}\}\}$	$\{\{a\}, \{end\}\}$	$\{(start, \{\{b\}\}), (a, \{\{a\}, \{end\}\})\}$
b	$\{\{\mathbf{start}\}\}$	$\{\{end\}\}$	$\{(a, \{\{a\}, \{end\}\}), (b, \{\{end\}\})\}$
a	$\{\{start\}, \{\mathbf{a}\}\}$	$\{\{a\}, \{end\}\}$	$\{(b, \{\{end\}\}), (a, \{\{a\}, \{end\}\})\}$
a	$\{\{start\}, \{\mathbf{a}\}\}$	$\{\{a\}, \{end\}\}$	$\{(b, \{\{end\}\}), (a, \{\{a\}, \{end\}\})\}$
a	$\{\{start\}, \{\mathbf{a}\}\}$	$\{\{a\}, \{end\}\}$	$\{(b, \{\{end\}\}), (a, \{\{a\}, \{end\}\})\}$
end	$\{\{\mathbf{a}, \mathbf{b}\}\}$	$\{\{\emptyset\}\}$	$\{\}$

Using the event-local execution semantics for Causal nets, various conformance checking measure can be defined. First of all, we can apply the Improved Continuous Semantics (ICS) measure to be used with our defined execution semantics. The actual definition of the ICS measure itself is equal to the one applied by Heuristics Miner and its variants, i.e. equal to the fitness measure $PF_{complete}$ as described by Alves de Medeiros (Alves de Medeiros, 2006). As we have defined event-local execution semantics, the possibility also exists to re-utilize existing conformance checking measures which depend only on such semantics (i.e. determining whether an activity in a trace can be parsed by the model or not). These measures can directly be applied to our proposed approach, since our defined execution semantics allow to determine for each $a \in T_L$, given a list of pending obligations, whether this activity can be executed fittingly or not. Finally, recall that the discovered Causal nets can be converted to Petri nets (van der Aalst et al., 2011), which allows for a multitude of other conformance checking measures available in literature to be applied.

5. Experimental Evaluation

We perform an experiment evaluation to benchmark the robustness and performance of our approach using 50 different event logs (see Table 1). Logs “a10skip” to “l2lskip” are commonly used *synthetic* event logs (Alves de Medeiros, 2006). Logs “prAm6” to “prGm6” are also synthetic and have

been utilized in a benchmarking study by Munoz-Gama et al. (Munoz-Gama et al., 2013). Next, logs "perml*X*a*Y*" contain all *permutations* (with repetition) of length X with number of activity types equal to Y (not including distinct start/end activities); the log size is hence Y^X. The best model for these logs is obviously a "flower model" which allows any sequence of activities, but these logs will be used to perform robustness checks by iterating over and mining each trace separately. Logs "randpms*X*d*Y*" are logs with size X generated from a *randomly* constructed process model[3] with depth Y. Logs "rands*A*l*B*m*C*a*D*" are also randomly generated, but purely by choosing random activities out of an activity alphabet with size D (not including distinct start/end activities) to construct A traces with mean length B and standard deviation C, i.e. not simulated from a (random) process model. Logs "real*X*" encompass four *real life* logs. Table 1 also provides an overview of the structural characteristics for the event logs included in the experiment.

For our experimental evaluation, we include Heuristics Miner (Weijters and Ribeiro, 2011), using the "Mine for a Heuristics Net using Heuristics Miner" plugin (*HM*) in ProM 6.6, as well as the Flexible Heuristics Miner, using the "Mine for a Causal Net using Heuristics Miner" plugin in ProM 6.6 (*FHM*). These are benchmarked against Fodina (*F*), with *FD* describing a configuration with duplicate task mining being enabled as well.

Using this setup, we first evaluate the robustness of our technique. One might expect that a process discovery algorithm would be able to return a perfectly fitting process model in case where the given event log only contains one single trace variant. Considering for a moment that duplicate activities could be mined, such a process model could indeed simply model the

[3]Using the Process Log Generator, see: `http://www.processmining.it/sw/plg`

Event Log	$\lvert T_L \rvert$	$\lvert L \rvert$	$\lvert \bigcup(L) \rvert$
a10skip	12	300	6
a12	14	300	5
a5	7	300	13
a6nfc	8	300	3
a7	9	300	14
a8	10	300	4
betasimplified	13	300	4
choice	12	300	16
driverslicense	9	2	2
driverslicenseloop	11	350	87
herbstfig3p4	12	32	32
herbstfig5p19	8	300	6
herbstfig6p18	7	300	153
herbstfig6p31	9	300	4
herbstfig6p36	12	300	2
herbstfig6p38	7	300	5
herbstfig6p41	16	300	12
l2l	6	300	10
l2loptional	6	300	9
l2lskip	6	300	8
prAm6	363	1200	1049
prBm6	317	1200	1126
prCm6	311	500	500
prDm6	429	1200	1200
prEm6	275	1200	1200
prFm6	299	1200	1200
prGm6	335	1200	1200

Event Log	$\lvert T_L \rvert$	$\lvert L \rvert$	$\lvert \bigcup(L) \rvert$
perml10a3	5	59049	59049
perml3a10	12	1000	1000
perml3a3	5	27	27
perml3a5	7	125	125
perml5a10	12	100000	100000
perml5a3	5	243	243
perml5a5	7	3125	3125
randpms10000d1	8	10000	2
randpms10000d2	16	10000	4724
randpms10000d3	38	10000	2906
randpms1000d1	7	1000	17
randpms1000d2	14	1000	12
randpms1000d3	51	1000	998
randpms100d1	9	100	3
randpms100d2	18	100	55
randpms100d3	20	100	54
rands10000l20m8a10	12	10000	10000
rands1000l10m4a5	7	1000	999
rands100l5m2a3	5	100	90
realdocman	70	12391	1411
realhospital	626	1143	981
realincman	18	24770	1174
realoutsourcing	7	276599	3151

Table 1: Structural log characteristics for event logs included in experimental setup.

sequence of events as they occur in the trace variant to obtain such a fitting model. Therefore, we perform a basic analysis where, for each event log, each trace is mined separately by the discovery algorithm under consideration, after which the trace is replayed on the mined model to verify whether a fitting model was constructed. An end score is then obtained for each log representing the percentage of traces for which such a fitting model could be mined. To replay each trace on its associated mined model, we apply each discovery algorithm's "native" replay semantics. For Heuristics Miner, we apply the replay semantics as utilized by the Improved Continuous Semantics (ICS) measure. For Causal Nets mined by Flexible Heuristics Miner, we align each trace on its "Flex net" (the implementation in ProM denotes Causal Nets mined by this miner as "flexible nets"; a plugin is available to

Robustness Analysis							
Event Log	*HM*	*FHM*	*F* and *FD*	**Event Log**	*HM*	*FHM*	*F* and *FD*
perml10a3	0.18	0.64	**1.00**	rands10000l20m8a10	0.23	0.79	**1.00**
perml5a10	0.97	**1.00**	**1.00**	rands1000l10m4a5	0.42	0.83	**1.00**
perml5a3	0.76	0.94	**1.00**	rands100l5m2a3	0.76	0.93	**1.00**
perml5a5	0.89	0.98	**1.00**	realdocman	**1.00**	**1.00**	**1.00**
randpms10000d2	0.91	**1.00**	**1.00**	realhospital	0.54	0.82	**1.00**
randpms10000d3	0.91	0.99	**1.00**	realincman	**1.00**	**1.00**	**1.00**
randpms1000d3	0.81	0.95	**1.00**	realoutsourcing	0.93	**1.00**	**1.00**
randpms100d2	0.90	**1.00**	**1.00**	driverslicenseloop	0.81	0.95	**1.00**
				herbstfig6p18	0.86	0.86	**1.00**

Table 2: Robustness results for the evaluated discovery algorithms. For each event log, each trace is mined separately using the lowest dependency thresholds possible for each miner, after which the trace is replayed on the mined model to verify whether a fitting model was constructed. An end score is then obtained for each log representing the percentage of traces for which such a fitting model could be mined. Event logs for which each miner was able to obtain a perfect (1.00) result are omitted.

replay event logs by means of alignment). For Fodina, we apply the heuristic replay semantics as described in Section 4.2). Table 2 lists the results of this operation. The results show that Fodina is able to mine all single traces correctly. Note also that we relied here on *the most relaxed configuration parameters regarding dependency thresholds for each miner*, i.e. all dependency thresholds were set to their lowest values (zero) for all miners.

Next, we execute a standard benchmark comparison where each miner is applied on the event log as a whole, after which recall and precision of the discovered models is assessed. To evaluate the discovered models in a fair manner, we first perform a conversion to a Petri net. Note that we have *modified the conversion procedure in the case of Heuristics Miner to ensure a correct conversion*. We then execute the following conformance checking measures: Behavioral Recall (Goedertier et al., 2009) (r_B), to evaluate fitness, and Behavioral Weighted Precision (vanden Broucke et al., 2014) (p_B^w) to evaluate precision. Both work on an event-granular level and hence allow for a robust comparison among the different miners. Both metrics are com-

bined using the F1 measure (the harmonic mean of precision and recall), which has been previously applied in a process mining context, see (Weerdt et al., 2011).

Table 3 lists the results of the benchmarking experiment. For 36 out of 50 event logs, Fodina is able to achieve the highest F1 result; for 8 logs, only a limited number of results could be obtained within the set time limit. Note that Fodina is able to obtain much higher recall results if guided by the end-user to do so (i.e. in the low-threshold configurations). The results for *herbstfig6p38* show an interesting case where Heuristics Miner is able to significantly outperform Fodina. Note that this result would not be achieved when using the default Petri net conversion available for Heuristics Miner (in which case the results drop significantly).

6. Conclusion

This paper has presented Fodina, a process discovery technique which follows the generic idea of heuristic process discovery algorithms. Although such techniques have proven themselves as robust process discovery algorithms and able to deal with real life event logs containing a large amount of variety of behavior, we identified some particular issues which limit the robustness and reliability of the technique. As such, we have set out to perform a thorough literature review and evaluation of the existing heuristic process discovery variants with their implementation to consequently propose a new technique which was proven to be more robust via a comprehensive evaluation experiment. Furthermore, the proposed technique presents various contributions, most notably the capability to mine duplicate tasks and the ability to configure various options to guide the discovery algorithms.

	Miner			
Event Log	*HM*	*FHM*	*F*	*FD*
perml10a3	**0.94** (1.00 0.89)	0.80 (0.82 0.78)	**0.94** (1.00 0.89)	**0.94** (1.00 0.89)
perml3a10	0.54 (0.80 0.41)	0.63 (0.72 0.56)	0.64 (0.72 0.57)	**0.65** (0.73 0.58)
perml3a3	0.75 (0.67 0.85)	0.74 (0.85 0.65)	**0.79** (0.73 0.87)	0.77 (0.69 0.87)
perml3a5	0.66 (0.80 0.56)	0.68 (0.78 0.60)	**0.72** (0.83 0.63)	**0.72** (0.82 0.64)
perml5a10	0.71 (0.86 0.61)	– (0.70 –)	**0.80** (0.67 1.00)	– (0.67 –)
perml5a3	**0.87** (0.95 0.81)	0.78 (0.87 0.71)	**0.87** (0.95 0.81)	**0.87** (0.95 0.81)
perml5a5	**0.79** (0.86 0.74)	0.73 (0.77 0.68)	**0.79** (0.85 0.73)	**0.79** (0.85 0.73)
randpms10000d1	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
randpms10000d2	**0.96** (1.00 0.92)	**0.96** (1.00 0.92)	**0.96** (1.00 0.92)	**0.96** (1.00 0.92)
randpms10000d3	**0.96** (1.00 0.93)	**0.96** (1.00 0.93)	**0.96** (1.00 0.93)	**0.96** (1.00 0.93)
randpms1000d1	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
randpms1000d2	**0.91** (1.00 0.83)	**0.91** (1.00 0.83)	**0.91** (1.00 0.83)	**0.91** (1.00 0.83)
randpms1000d3	**0.83** (1.00 0.71)	**0.83** (1.00 0.71)	**0.83** (1.00 0.71)	**0.83** (1.00 0.71)
randpms100d1	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
randpms100d2	**0.94** (1.00 0.89)	**0.94** (1.00 0.89)	**0.94** (1.00 0.89)	**0.94** (1.00 0.89)
randpms100d3	**0.95** (1.00 0.90)	**0.95** (1.00 0.90)	**0.95** (1.00 0.90)	**0.95** (1.00 0.90)
rands10000l20m8a10	**0.29** (0.95 0.17)	– (0.67 –)	0.28 (0.87 0.17)	0.28 (0.88 0.17)
rands1000l10m4a5	**0.59** (0.91 0.44)	0.55 (0.77 0.43)	0.57 (0.90 0.42)	0.58 (0.93 0.43)
rands100l5m2a3	0.80 (0.92 0.71)	0.76 (0.88 0.67)	**0.81** (0.97 0.70)	0.79 (0.90 0.70)
realdocman	–	–	**0.56** (0.97 0.39)	**0.56** (0.97 0.39)
realhospital	–	–	–	–
realincman	0.53 (0.80 0.40)	0.70 (0.95 0.56)	0.80 (0.96 0.68)	**0.81** (0.97 0.70)
realoutsourcing	0.07 (0.04 0.62)	0.82 (0.76 0.90)	**0.97** (1.00 0.95)	**0.97** (1.00 0.95)
a10skip	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
a12	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
a5	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
a6nfc	0.93 (1.00 0.87)	**0.96** (1.00 0.92)	0.89 (0.99 0.81)	0.89 (0.99 0.81)
a7	0.89 (1.00 0.80)	0.87 (0.98 0.77)	**0.95** (0.94 0.95)	**0.94** (0.94 0.94)
a8	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
betasimplified	**0.92** (1.00 0.85)	**0.92** (1.00 0.85)	**0.92** (1.00 0.85)	**0.92** (1.00 0.85)
choice	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
driverslicense	**0.95** (1.00 0.90)	**0.95** (1.00 0.90)	**0.95** (1.00 0.90)	**0.95** (1.00 0.90)
driverslicenseloop	**0.94** (1.00 0.89)	**0.94** (1.00 0.89)	**0.94** (1.00 0.89)	**0.94** (1.00 0.89)
herbstfig3p4	**1.00** (1.00 0.99)	**1.00** (1.00 0.99)	0.99 (1.00 0.98)	0.99 (1.00 0.98)
herbstfig5p19	**0.95** (1.00 0.90)	**0.95** (1.00 0.90)	**0.95** (1.00 0.90)	**0.95** (1.00 0.90)
herbstfig6p18	**0.99** (1.00 0.97)	**0.99** (1.00 0.97)	**0.99** (1.00 0.97)	**0.99** (1.00 0.97)
herbstfig6p31	**0.72** (1.00 0.56)	**0.72** (1.00 0.56)	**0.72** (1.00 0.56)	**0.72** (1.00 0.56)
herbstfig6p36	**0.99** (1.00 0.98)	**0.99** (1.00 0.98)	**0.99** (1.00 0.98)	**0.99** (1.00 0.98)
herbstfig6p38	**0.86** (0.88 0.84)	0.80 (1.00 0.66)	0.74 (0.96 0.61)	0.74 (0.96 0.61)
herbstfig6p41	– (1.00 –)	– (1.00 –)	– (1.00 –)	– (1.00 –)
l2l	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
l2loptional	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
l2lskip	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)	**1.00** (1.00 1.00)
prAm6	– (0.94 –)	– (0.93 –)	– (0.95 –)	– (0.95 –)
prBm6	**0.61** (0.98 0.44)	– (0.97 –)	– (0.96 –)	– (0.96 –)
prCm6	**0.07** (0.61 0.03)	– (0.57 –)	– (0.68 –)	– (0.69 –)
prDm6	– (0.54 –)	–	– (0.60 –)	– (0.59 –)
prEm6	**0.08** (0.73 0.04)	– (0.77 –)	– (0.76 –)	– (0.78 –)
prFm6	– (0.78 –)	– (0.78 –)	– (0.77 –)	– (0.76 –)
prGm6	– (0.65 –)	– (0.74 –)	– (0.73 –)	– (0.72 –)

Table 3: F1-measure results for the evaluated discovery algorithms. Recall and precision scores are reported between parentheses. "–" results correspond with cases where the conformance checking procedure took too much time (more than two hours) and was aborted.

References

Alves de Medeiros, A., 2006. Genetic process mining. Ph.D. thesis, TU Eindhoven.

Alves de Medeiros, A., Weijters, A., van der Aalst, W., 2007. Genetic process

mining: an experimental evaluation. Data Min. Knowl. Discov. 14 (2), 245–304.

Burattin, A., Sperduti, A., 2010. Heuristics miner for time intervals. In: ESANN.

Burattin, A., Sperduti, A., van der Aalst, W., 2012. Heuristics miners for streaming event data. CoRR abs/1212.6383.

De Weerdt, J., De Backer, M., Vanthienen, J., Baesens, B., 2012. A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Information Systems 37 (7), 654–676.

Goedertier, S., Martens, D., Vanthienen, J., Baesens, B., 2009. Robust process discovery with artificial negative events. Journal of Machine Learning Research 10, 1305–1340.

Günther, C., 2009. Process mining in flexible environments. Ph.D. thesis, TU Eindhoven.

Lu, X., Fahland, D., van den Biggelaar, F. J. H. M., van der Aalst, W. M. P., 2016. Handling Duplicated Tasks in Process Discovery by Refining Event Labels. Springer International Publishing, Cham, pp. 90–107.

Maruster, L., Weijters, A., van der Aalst, W., van den Bosch, A., 2006. A rule-based approach for process discovery: Dealing with noise and imbalance in process logs. Data Mining and Knowledge Discovery 13 (1), 67–87.

Munoz-Gama, J., Carmona, J., van der Aalst, W., 2013. Conformance checking in the large: Partitioning and topology. In: Daniel, F., Wang, J., Weber, B. (Eds.), BPM. Vol. 8094 of Lecture Notes in Computer Science. Springer, pp. 130–145.

van der Aalst, W., 2011. Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer.

van der Aalst, W., Adriansyah, A., van Dongen, B., 2011. Causal nets: A modeling language tailored towards process discovery. In: Katoen, J., König, B. (Eds.), CONCUR. Vol. 6901 of Lecture Notes in Computer Science. Springer, pp. 28–42.

van der Aalst, W., Weijters, A., Maruster, L., 2004. Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. 16 (9), 1128–1142.

vanden Broucke, S. K. L. M., Weerdt, J. D., Vanthienen, J., Baesens, B., 2014. Determining process model precision and generalization with weighted artificial negative events. IEEE Transactions on Knowledge and Data Engineering 26 (8), 1877–1889.

Weerdt, J. D., Backer, M. D., Vanthienen, J., Baesens, B., 2011. A robust f-measure for evaluating discovered process models. In: 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM). pp. 148–155.

Weijters, A., Ribeiro, J., 2011. Flexible heuristics miner (fhm). In: CIDM. IEEE, pp. 310–317.

Weijters, A., van der Aalst, W., Alves de Medeiros, A., 2006. Process mining with the heuristicsminer algorithm. BETA working paper series 166, TU Eindhoven.