Discovering an S-Coverable WF-net using DiSCover

Eric Verbeek[0000-0002-1658-9679]
Department of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
h.m.w.verbeek@tue.nl

***Abstract*—Although many algorithms exist that can discover a WF-net from an event log, only a few (if any at all) can discover advanced routing constructs. As examples, the Inductive miner uses process trees and cannot discover complex loops, or situations where choice and parallel behavior is mixed, and the Hybrid ILP miner cannot discover certain complex routing constructs because it cannot discover silent transitions. This paper introduces the DiSCover miner, a discovery algorithm that can discover these more complex constructs and that is implemented in ProM. The DiSCover miner discovers from the event log a WF-net that corresponds to a collection of state machines that need to synchronize on the visible transitions (that is, on the activities from the event log). As such, it discovers a WF-net that is S-Coverable but not necessarily sound. Initial results show that it can discover complex routing constructs and that it performs well on the data sets of the different Process Discovery Contests. It even preformed better than winners of the 2020 and 2021 contests.**

***Index Terms*—event log, discovery, S-coverable, WF-net, ProM, Process Discovery Contest**

I. Introduction

Process discovery has been around now for about twenty years. In these twenty years, a good number of different miners have been introduced. Some of these miners result in imperative models, like a workflow net (WF-net), a BPMN diagram, or a process tree, others result in declarative models like Declare, DCR graphs, and log skeletons. In this paper, we will mainly consider the miners that result in imperative models, and we assume that imperative models in general can be captured by a WF-net.

The miners in the commercial tools typically use a simple strategy based on the so-called directly-follows graph. This graph contains information how often activities were executed, and how often they directly follow each other. This can be computed from an event log in a straightforward manner, and comes with the guarantee that the entire event log can be replayed perfectly on it. To avoid too much clutter in the graphs, these tools often allow the user to filter out infrequent nodes and/or infrequent edges. An advantage of this filtering is that the graph may become more insightful to the user. A disadvantage is that the event log can not be replayed perfectly on the filtered graph. As the directly-follows graph is basically a state machine, it does not allow for any concurrency. As a result, it is obvious that these tools cannot discover concurrency.

This work made use of the Dutch national e-infrastructure with the support of the SURF Cooperative using grant no. EINF-3334.

The academic tools that use imperative models have been trying to fill this gap in different ways. Some miners are still based on the directly-follows graph, but use this graph to decide which activities are concurrent. The Alpha miner [1] is an example of such an algorithm. If two activities can directly follow each other, then they are considered to be concurrent, and no path of directed arcs will connect them. Other miners that also use the directly-follows graph use process trees as the underlying formalism. The Inductive miner [2] is an example of such a miner. This miner works by finding specific cuts in the directly-follows graph. If an appropriate cut is found, the graph is split accordingly, and the event log as well. This procedure is then repeated on the event logs that result from the split, until the event logs have become trivial (like a single activity is left). The downside of using process trees is that more complex structures cannot be captured completely. As a result, the Inductive miner can only discover a limited collection of WF-nets.

Other academic tools do not use the directly-follows graph. Examples include region-based miners, like the Hybrid ILP miner [3], the Prime miner [4], and the eST miner [5]. Both the Hybrid ILP miner and the eST miner start with introducing a transition for every unique activity in the event log, and then add places that do not restrict any trace in some way (this differs per miner). Although thees miners can result in WF-nets with complex structures, they cannot introduce any silent transitions in these WF-nets, which limits what they can do. As a result, they are limited to whatever it can do with the transitions that correspond directly to an activity. For example skipping an activity is hard to model if you cannot use a silent transition that models this skip. The Prime miner potentially results in very good WF-nets. Basically, it clusters traces into event structures, then discovers a WF-net for every event structure, and finally merges everything into a single WF-net. Unfortunately, in practice this miner often fails to produce a result in reasonable time because the merging step simply takes too long.

For all of these miners it holds that they could not successfully compete in the Process Discovery Contests[1] (PDCs), which is a series of contests for process discovery algorithms. As examples, the PDC of 2020 was won by a miner that returned a WF-net that resulted from filtering the directly-follows graph, while the PDC of 2021 was won by a miner

[1]See https://www.tf-pm.org/competitions-awards/discovery-contest

that resulted in DCR graphs [6], that is, in declarative models.

This paper offers an approach that aims to extend the commercial miners with a simple and flexible way to detect concurrency, such that the resulting miner is competitive in the PDCs. Basically, the approach splits the directly-follows graph into a collection of smaller directly-follows graphs containing only activities that do not exhibit concurrency. Using an algorithm similar to the miners in the commercial tools, a collection of state machines is then discovered from these smaller graphs. By merging these discovered state machines on the activities, concurrency is finally added to the discovered WF-net. Because the final WF-net is constructed by merging transitions of state machine WF-nets, it is S-Coverable [7] by construction.

The remainder of this paper is organized as follows. Section II first introduces some preliminaries, after which Section III introduces the proposed DiSCover miner. Section IV introduces the implementation of the miner in the ProM 6.12 release. Section V puts this implementation to the test on the data sets of the Process Discovery Contests. Finally, Section VI concludes the paper.

II. Preliminaries

A. Event logs

We consider an event log to be a multi-set (or bag) of sequences over some alphabet of activities. Sometimes, this restricted form of event logs is also referred to as activity logs, as they only contain the activities and no other attributes (like information on when the activity was executed or on the resource that executed the activity).

As the running example for an event log, we use the following event log, which was taken from [8]: Example event log $L = [\langle a,b,c,g,e,h\rangle^{10}, \langle a,b,c,f,g,h\rangle^{10}, \langle a,b,d,g,e,h\rangle^{10}, \langle a,b,d,e,g,h\rangle^{10}, \langle a,b,e,c,g,h\rangle^{10}, \langle a,b,e,d,g,h\rangle^{10}, \langle a,c,b,e,g,h\rangle^{10}, \langle a,c,b,f,g,h\rangle^{10}, \langle a,d,b,e,g,h\rangle^{10}, \langle a,d,b,f,g,h\rangle^{10}]$. As an example, in this event log, the activity sequence where a, b, c, g, e, and h are executed in that order was recorded 10 times.

In the example event log, activities e and f appear to be in a choice, as precisely one of them occurs in any trace. However, where activity f always precedes g, e can either precede or follow g. As such, e seems to be concurrent to g, but f seems to be in a sequence with it. There also seems to be choice between c and d, but where d can both directly follow e and directly precede it, c can only directly follow it. Such mixtures of choice and concurrency are typically hard to discover by existing miners.

To make the start and end of traces explicit, we can implicitly extend the traces with artificial start and end activities. As an example, the trace $\langle a,b,c,g,e,h\rangle$ would then become $\langle \blacktriangleright, a,b,c,g,e,h,\blacksquare\rangle$, where $\blacktriangleright$ is the artificial start activity and $\blacksquare$ is the artificial end activity. This paper assumes that these artificial activities are present in any event log, either explicitly or implicitly.

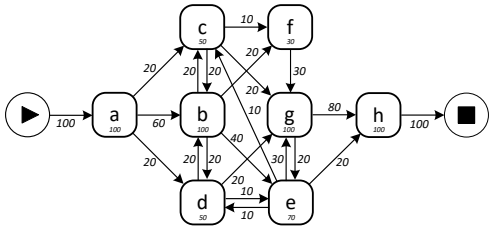

Fig. 1. Directly-follows graph G for the event log L.

B. Directly-follows graph

Fig. 1 shows the directly-follows graph for the example event log. This graph shows how often activities were executed in the event log, but also how often a first activity was directly followed by a second activity. As examples, activity e was executed 70 times, and it was 10 times directly followed by c, 10 times by d, 30 times by g, and 20 times by h.

C. Commercial miners

Many commercial miners are based on this directly-follows graph, and many of them (including Fluxicon's Disco and Celonis' miner) result in a directly-follows graph with some filtering applied to it. As a result, these miners cannot discover concurrency, instead they discover loops, like shown in Fig. 1. As an example, the back-and-forth arcs between activities b and d suggest that these activities can be executed in any order, and not that they can be executed arbitrarily many times.

The advantage of discovering a directly-follows graph is that it is very simple and can be done very fast. We only need to traverse the event log once, while we keep count of how many times activities were executed and how many times they followed each other directly. Filtering the directly-follows graph may be more involved, and may require sophisticated heuristics, but still may be very fast. As a result, discovering and showing a filtered directly-follows graph is fast, which explains why it is used in many commercial tools.

D. Academic miners

Academic tools typically try to detect concurrency. As a result, these miners are usually less simple, and less fast. Many of the existing miners aim to discover a WF-net [9], or a process model that can be easily converted into a WF-net. A WF-net is a Petri net [10]–[12] with a single source place (no incoming arcs) and a single sink place (no outgoing arcs). The source place is usually labeled i and the sink place is usually labeled o. An important property of a WF-net is whether it is sound. If a WF-net is sound, the desired final state (a single token in place o) can always be reached from the initial state (a single token in place i), and any transition can be executed.

Some of these miners do start with discovering a directly-follows graph, and then try to detect concurrency from this graph. Examples of these discoverers include the Alpha miner [1] and the Inductive miner [2].

Fig. 2 shows the WF-net that is discovered by the Alpha miner from the directly-follows graph G. This WF-net has

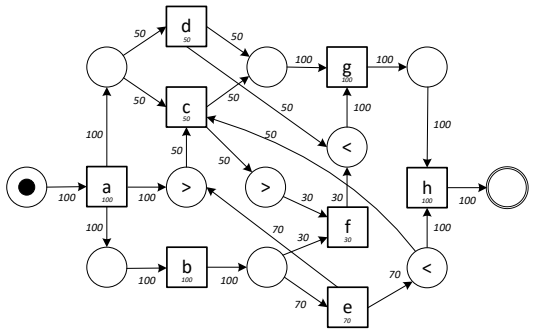

Fig. 2. WF-net discovered from the directly-follows graph G by the alpha miner.

been extended with similar information as G about how many times activities were executed, and how often they directly followed each other. Clearly, these numbers do not match for the four places indicated with $>$ or $<$: Either more tokens are produced for such a place then are consumed ($>$), or more are consumed then are produced ($<$). As a result, it will be hard (if possible at all) to replay the event log correctly on this discovered WF-net.

However, note that many of the complex features of the event log are present in the WF-net as discovered by the Alpha miner:

- Activity b occurs concurrent with c and d.
- There is a mandatory choice between c and d.
- There is a mandatory choice between e and f.
- If e occurs, it occurs concurrent with g.
- If f occurs, it occurs before g.

The problem for the Alpha miner is that it does not have silent transitions (that is, transitions not directly corresponding to any activity in the event log) to glue these discovered features nicely together.

Fig. 3 shows the process tree [13] discovered by the Inductive miner from the directly-follows graph G. Clearly, this process tree allows for both activity e and f to occur, or that neither of them occur, and it also allows f to occur after g has occurred. As a result this process tree does not capture some of the complex features of the event log.

III. The DiSCover miner

This section introduces the DiSCover miner, which takes the complex features of the event log into account in such a way that the event log can still be perfectly replayed on the discovered WF-net. First, we consider the situation where the event log does not contain any noise, and we will show that the discovered WF-net can then perfectly replay the event log. Second, we add noise into the picture by dropping this assumption that the event log does not contain any noise.

A. Assuming no noise

Like the Alpha miner and the Inductive miner, the DiSCover miner starts from the directly-follows graph G. However, unlike these other miners, the DiSCover miner tackles the issue

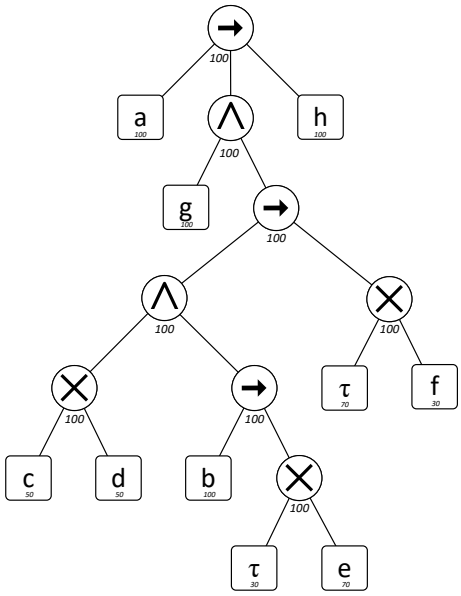

Fig. 3. Process tree discovered from the directly-follows graph G by the inductive miner.

of concurrency by detecting maximal subsets of activities that do not exhibit concurrency, that is, that do not have these back-and-forth edges between them. For graph G, four such a maximal subsets of activities can be detected: $\{a, b, e, f, h\}$, $\{a, b, f, g, h\}$, $\{a, c, d, f, g, h\}$, and $\{a, c, e, f, h\}$.

For each of these maximal subsets, the DiSCover miner then projects the event log on this subset and discovers a directly-follows graph for it. As an example, Fig. 4 shows the directly-follows graph for the subset $\{a, c, e, f, h\}$. As we assume that the activities in the subset are not concurrent, we can simply create a state machine WF-net from this directly-follows graph (like the commercial tools do). As a result, the DiSCover miner discovers a state machine WF-net for every maximal subset of non-concurrent activities.

Fig. 5 shows the WF-net that is constructed by merging (on the artificial start and end activities (▶ and ■)) the state machine WF-nets that are discovered for all four subsets. The top-most branch in this WF-net corresponds to the state machine WF-net discovered for the subset $\{a, c, e, f, h\}$, the subsets for the other three branches can be detected easily.

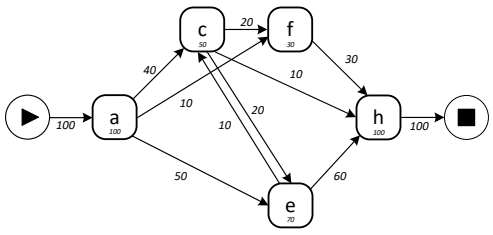

Fig. 4. Directly-follows graph for the example event log when projected onto the activities a, c, e, f, and h.

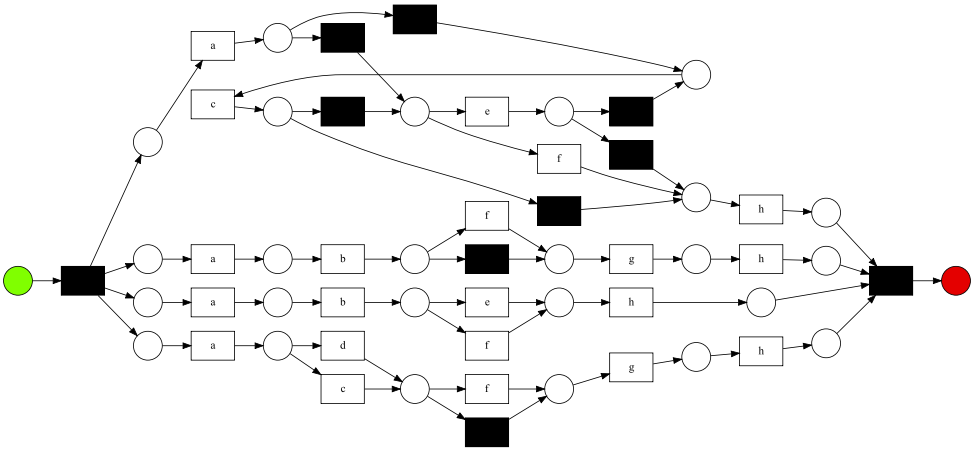

Fig. 5. WF-net showing discovered state machines for the example event log L. The top state machine corresponds to the directly-follows graph shown in Fig. 4.

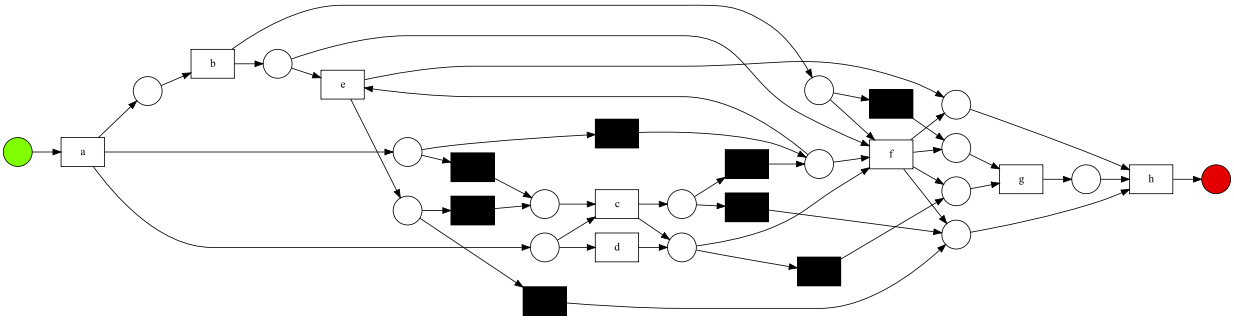

Fig. 6. WF-net N discovered from the example event log L.

The only thing we now need to do, is to merge all transitions that correspond to the same activity, and we obtain a WF-net that is S-coverable [7] by definition. This merging step can potentially be followed by a reduction step that uses well-known reduction rules that preserve the behavior of the WF-net [14], [15]. Fig. 6 shows the resulting WF-net N.

It is straightforward to show that the WF-net N can successfully replay all the traces from the event log L, as the merged net can only block on the next activity in the trace if one of the contained state machines would block on it. But by construction, a state machine cannot block on the next activity as it was constructed to allow for all traces. As a result, the DiSCover miner now has discovered a WF-net that contains complex routing constructs that can perfectly replay the event log it was discovered from. However, this net may not be sound. As an example, after transitions a and c (and the silent transition that connects them) have occurred, the bottom-most of the two silent transitions that follow transition c can now occur. This effectively prevents transitions e and f from occurring, while one of them needs to occur to reach completion (cf. the state machine for $\{a, b, e, f, h\}$ in Fig. 5).

B. Adding noise

Noise in the event log can be taken into account by filtering the directly-follows graph prior to using it for (i) detecting concurrency (for the directly-follows graph of the original event log) or (ii) discovering a state machine WF-net (for the directly-follows graphs of the projected event logs). Basically, any filtering technique could work here, but problems may arise when the filtering removes paths needed to reach the final state from the initial state. Also note that, when filtering the directly-follows graph, the resulting WF-net may not be able to replay the original event log perfectly.

At the moment, the DiSCover miner comes with three thresholds that can be used to filter the directly-follows graph: an absolute threshold, a relative threshold, and a safety threshold. Let (s, t) be an edge in the directly-follows graph from a source activity s to a target activity t, and let w be the weight of this edge.

Absolute threshold τ_a
: The edge (s, t) is removed if $w \leq \tau_a$.

Relative threshold τ_r and safety threshold τ_s
: Let w_s be the maximal weight of any edge having s as source activity, and let w_t be the weight of any edge having t as target activity. The edge (s, t) is removed if (1) $100 \cdot w < \tau_s \cdot w_s \wedge 100 \cdot w < \tau_s \cdot w_t$ and (2) $100 \cdot w \leq \tau_r \cdot w_s \vee 100 \cdot w \leq \tau_r \cdot w_t$.

Using the absolute threshold, we can filter out lightweight edges. Using the relative threshold, we can filter out edges

that are lightweights in the set of edges that have s as the source activity or t as the target activity, *but* using the safety threshold, we can prevent filtering out some edges if they are heavyweights in either the set of edges that have s as source activity or in the set of edges that have t as target activity. As an example, let $\tau_r = 80$, $\tau_s = 95$, $w = 0.99 \cdot w_s$ and $w = 0.1 \cdot w_t$. As $100 \cdot w > 95 \cdot w_s$, this edge will not be removed because it is a heavyweight in the set of edges that have s as source activity.

IV. The DiSCover implementation

The DiSCover miner has been implemented in the DiSCover package in the ProM 6.12 release[2]. This package contains three different ProM DiSCover plugins:

DiSCover Petri net (user)
: Allows the user to select the thresholds, see also Fig. 7. Apart from these thresholds, the user can also limit the number of components, whether to merge the transitions that correspond to the activities, whether to reduce the merged Petri net, and whether to use a veto-ing scheme for noise. In a veto-ing scheme, an edge is only considered to be noise if all the components agree on it being noise. Table I shows the defaults values for these settings. The miner then discovers the WF-net using these settings. This plugin was used on the example event log with absolute threshold and noise level both 0 to discover the WF-nets shown by Fig. 5 and Fig. 6.

DiSCover Petri net (provided)
: Discovers the WF-net using the provided settings.

DiSCover Petri net (last)
: Discovers the WF-net using the settings as were last used by any of the three plugins. This allows the user to quickly rerun the DiSCover plugin using the same settings on another event log.

The main plugin uses the following nine steps to discover a WF-net from an event log:

1) The directly-follows graph is constructed from the event log.

[2]See https://www.promtools.org/

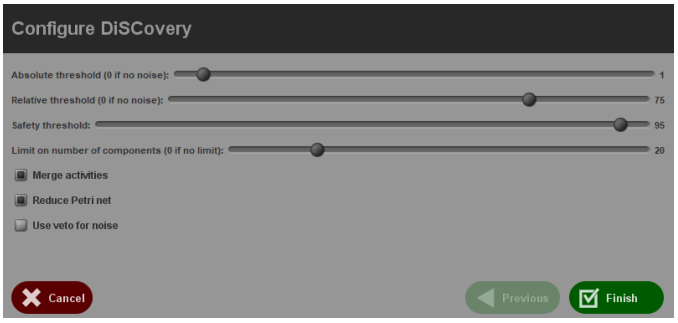

Fig. 7. The dialog for the main DiSCover plugin as implemented in ProM 6.12.

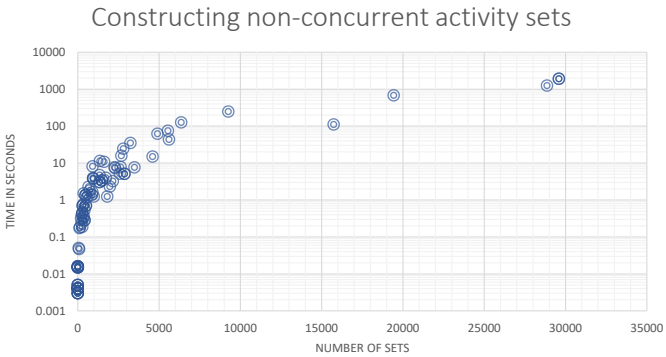

Fig. 8. Computation times for the non-concurrent activity sets.

2) A list of concurrent activity pairs is derived from this directly-follows graph, where two activities are presumed to be concurrent if they have back-and-forth edges in the graph.
3) A collection of maximal non-concurrent activity sets is derived from this list. This may take considerable time, as we have seen cases where more than 10,000 different sets need to be derived.
4) For every such set, a sublog is derived by filtering the event log on the set of activities.
5) For every such sublog, a directly-follows graph is constructed.
6) The number of these subgraphs may be reduced to a level that can be managed, as generating and merging more than 1000 WF-nets seems not a good idea.
7) For every remaining subgraph, a state-machine WF-net is constructed in a straightforward way.
8) The state-machine WF-nets are merged into a single WF-net is a straightforward way.
9) The merged WF-net is reduced as much as possible using existing or dedicated reduction rules. Especially if the WF-net contains many places and many transitions, this may take considerable time.

Noise filtering is applied to the directly follows graph as constructed in step 1 as well as to the directly-follows graphs constructed in step 5. Note that the same thresholds are used for the filtering in both steps.

Step 3 may take considerable time. Fig. 8 shows the case we encountered with the most extreme computation times. This

TABLE I
DEFAULT VALUES FOR DISCOVER PLUGIN SETTINGS

Setting	Default value
Absolute threshold	1
Relative threshold	1
Safety threshold	95
Number of components	20
Merge activities	Yes
Reduce Petri net	Yes
Use veto for noise	No

shows that it takes less than 10 seconds to construct up to 5000 different sets, but that it may take considerable more time if there are many more sets to be constructed. In the extreme case, the plugin had to construct 29,600 sets, which took about 1900 seconds.

To reduce the number of subgraphs in step 6, the plugin first uses an ILP to construct a minimal set of subgraphs such that all possible input sets and all possible output sets are still covered. Second, if the number of subgraphs still exceeds the number of components as set by the user, then we just take the required number of subgraphs.

The construction of the state-machine WF-net in step 7 uses the following steps:

1) For every node in the directly-follows graph, that is, for every activity, a transition is added which has the activity as its label. If the activity is artificial, then the transition will be silent, otherwise, it will be visible.
2) For every unique input set of any node, an input place is created. The input set of a node is the set of activities that can directly precede that node. Note that for the artificial start activity the input set is the empty set.
3) For every unique output set of any node, an output place is created. The output set of a node is the set of activities that can directly follow that node. Note that for the artificial end activity the output set is the empty set.
4) An arc is added from an input place to a transition if the corresponding node has the corresponding input set.
5) An arc is added from a transition to an output place if the corresponding node has the corresponding output set.
6) For every edge in the directly-follows graph, a silent transition and two arcs are added:
    a) An arc from the output place of the source node to the silent transition.
    b) An arc from the silent transition to the input place of the target node.

This way, by construction, every path through the WF-net is a sequence of triplets (input place, transition, output place) concatenated by silent transitions. As an example, Fig. 9 shows the four state-machine WF-nets (combined in a single WF-net by merging on the artificial start and end activities) constructed for the example event log L. Note that applying the reduction rules on this WF-net results in the WF-net shown in Fig. 5.

Fig. 10 shows the WF-net that results from this plugin using as input the BPIC 2012 event log [16] that has been filtered on only the *A*- and *O*-activities and using the values as shown by Fig. 7. Although this WF-net is certainly not sound (consider, for example, the *O_DECLINED+COMPLETE* activity), and can be improved on (for example, the four silent transitions to the left and right of the *A_FINALIZED+COMPLETE* can be merged, after which some places can be merged as well), this WF-net does provide some insights on the process, like that there is a loop containing activities related to an offer (the *O*-activities).

V. Putting it to the test

To put the DiSCover miner to the test, we have tested it on the data sets [17]–[21] of the different Process Discovery Contests, including the data set of the Process Discovery Contest of 2022[3]. For sake of completeness, we mention that we used a dedicated token-based replay algorithm for classifying traces on the discovered nets, and that this algorithm has been implemented as the "Classify DiSCovered Petri net" plugin[4]. The reason for using a token-based replay algorithm over an alignment-based replay algorithm was that the alignment-based replay algorithm resulted in too many timeouts (especially for the 2021 contest). The token-based replay algorithm is fast and is deterministic until the first mismatch occurs, after which it may become nondeterministic. To alleviate this issue, we have run all tests three times like is usual for the Process Discovery Contests of 2020, 2021, and 2022, and report only the numbers on which all three runs agree.

Table II shows the results. The first line shows the perfect score for every contest, the second line shows the winning score, and the additional lines show the result given an absolute threshold τ_a and a relative threshold τ_r (default values were used for all other settings, see Table I). The best scores for a contest are highlighted using a bold face, which shows that the DiSCover plugin performs better than the winning submissions of 2020 and 2021.

Note that the DiSCover plugin did not finish (*DNF*) on the 2021 data set when setting both thresholds to 0. This was caused by the fact that the ILP reduction did not finish for the event logs for which (roughly speaking) 15,000 or more subgraphs were constructed. In a next release of the plugin, we plan to make this ILP-based reduction optional to avoid this problem.

The best score of the DiSCover plugin on the PDC 2022 data set is 88.5%, which is just below the overall score of 90.0% of the Trace miner. This is acceptable, because the size of models obtained using the DiSCover miner is way less than the size as obtained using the Trace miner. As an example, for the most complex scenario (long-term dependencies, complex loops, OR constructs, routing constructs, optional tasks, duplicate tasks, heterogeneous noise (40% removed, 20% moved,

[3]See https://icpmconference.org/2022/process-discovery-contest/

[4]This plugin is also contained in the DiSCover package in the ProM 6.12 release.

TABLE II
Results on the different PDC data sets

PDC Year		**2016**	**2017**	**2019**	**2020**	**2021**	**2022**
Perfect		200	200	900	100.0%	100.0%	100.0%
Winner		**193**	**197**	**898**	76.2%	96.2%	*TBD*
$\tau_\mathbf{a}$	$\tau_\mathbf{r}$						
1	10	145	160	649	31.2%	76.0%	87.2%
1	5	154	163	728	55.9%	92.1%	88.5%
1	2	173	171	795	76.8%	97.6%	87.3%
1	1	188	170	819	**86.5%**	**98.2%**	83.8%
0	0	190	169	845	67.8%	*DNF*	66.6%

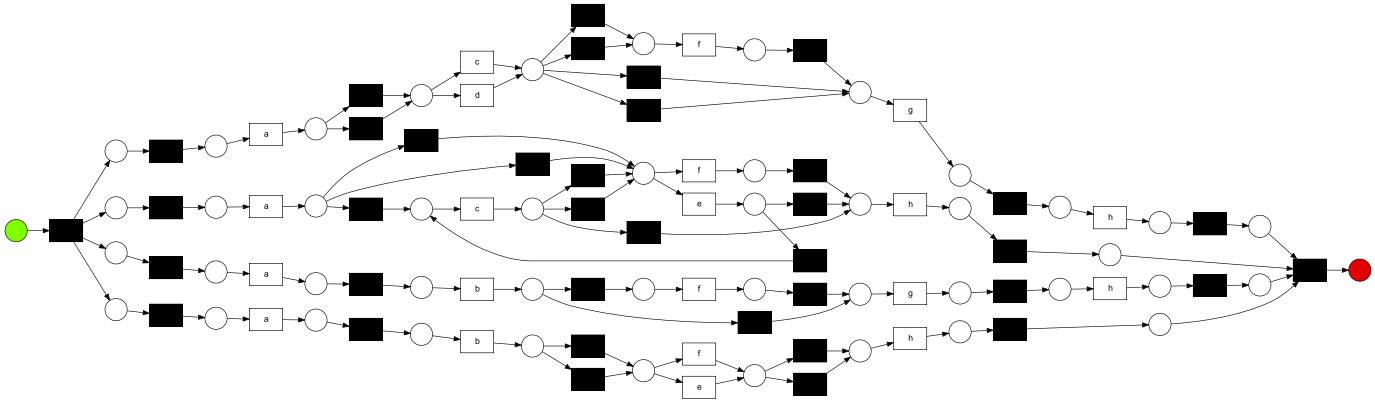

Fig. 9. State-machine WF-nets discovered from the example event log L: In-between two visible transitions, there is always a silent transition. Fig. 6 shows this WF-net after merging all visible transitions and applying all reduction rules, whereas Fig. 5 shows this WF-net after only applying all reduction rules.

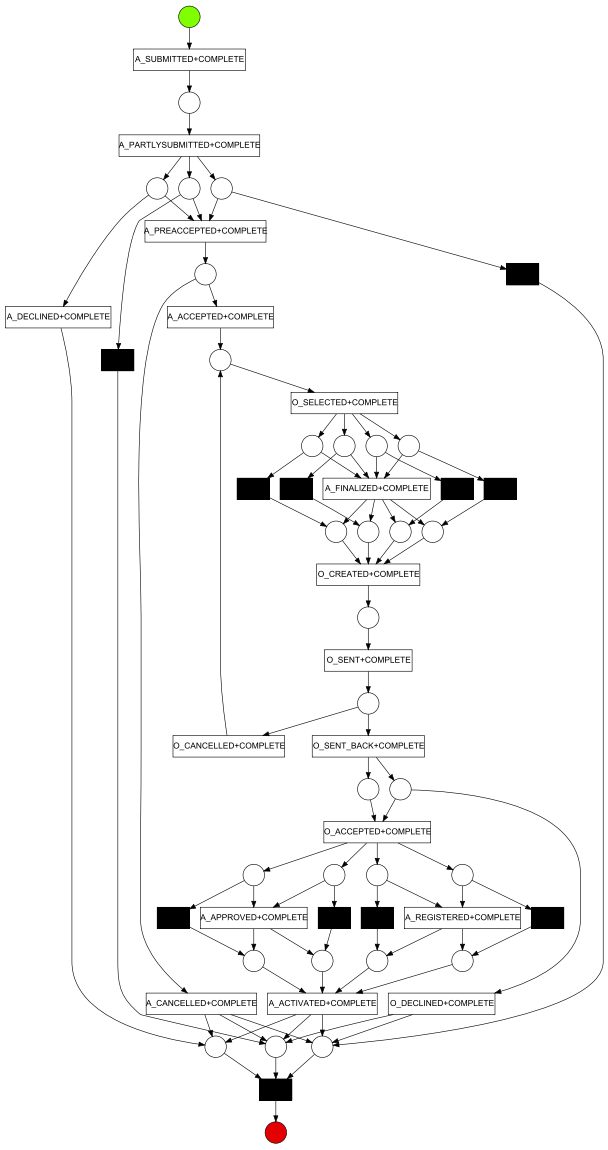

Fig. 10. The WF-net discovered using the plugin from the BPIC 2012 event log filtered on only the *A*- and *O*-activities.

and 40% copied), the WF-net discovered by the DiSCover miner (shown in Fig. 11) contains 30 places, 64 transitions and 148 arcs, whereas the WF-net discovered by the Trace miner contains 600 places, 1004 transitions and 2006 arcs (shown in Fig. 12). Although both WF-nets are too complex to see

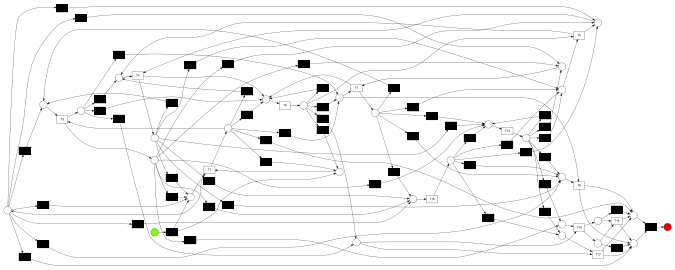

Fig. 11. Resulting WF-net of the DiSCover miner on the most complex scenario of the PDC 2022 data set: Somewhat readable, offers some insights.

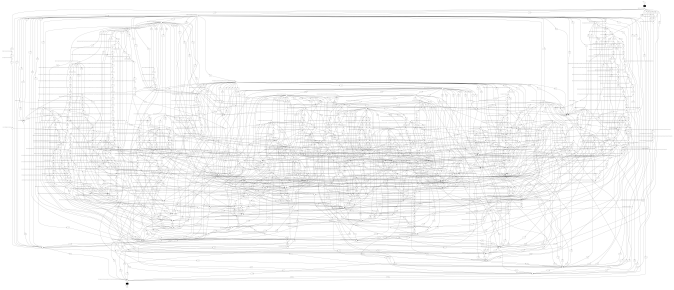

Fig. 12. Resulting WF-net of the Trace miner on the most complex scenario of the PDC 2022 data set: Unreadable, offers no insights.

(m)any details, it is clear that the WF-net discovered by the DiSCover miner is better readable and provides more insights than the WF-net discovered by the Trace miner.

VI. Conclusion

This paper has introduced a miner that potentially can add concurrency to commercial miners. Instead of discovering a single state machine WF-net from the original directly-follows graph, multiple state machine WF-nets are discovered

by this miner from multiple directly-follows graphs. The idea here is that for each of these multiple directly-follows graphs its activities do not exhibit concurrency anymore. As a result, discovering a state machine WF-net from such a directly-follows graph makes perfect sense. In a final step, the discovered state machine WF-nets are merged into a single WF-net, which brings concurrency back into the picture.

The most complex step in this approach is the generation of the multiple directly-follows graphs, as this is searching for maximal subsets of activities that are pairwise not concurrent. Also, applying existing behavior-preserving reduction rules on the discovered state machines and the merged WF-net may take some time.

The approach has been implemented as a plugin in ProM 6.12, which has been tested on the data sets of the different Process Discovery Contests. Results show that the DiSCover miner is very competitive in these contests, and that it even could have won the 2020 and 2021 editions of this contest.

An interesting idea for future work could be to add a "concurrency slider" to the DiSCover miner. If the slider is set to its lowest value, then the miner returns a state machine WF-net that is generated from the original directly-follows graph. If the slider is increased, the miners starts creating multiple directly-follows graphs. As an example, it could have a look at the two activities with back-and-forth edges that have the largest combined weight in any directly-follows graph, and replace this graph with the two directly-follows graphs that result by removing one of these activities. The subsets covered by these graphs may still exhibit some concurrency, but the more the slider is increased, the more directly-follows graphs are used, and in the end each of these directly-follows graphs contains activities that are not concurrent. This way, by increasing the slider, we increase the level of concurrency in the resulting model.

Some readers may actually prefer the WF-net where the different state machine WF-nets have not been merged over the WF-net where they have been merged, as the former WF-net may provide more insights. As an example, from Fig. 5 it is straightforward to see that either activity e or activity f has to occur, whereas from Fig. 6 this is not immediately clear. This begs the question whether we need to merge the state machine WF-nets at all. We could simply change the semantics of the WF-nets in such a way that all transitions that correspond to the same activity should occur synchronously: If one occurs, then all others need to occur as well. This may result in WF-nets that offer more insights to the user.

A final idea for future work is to create dedicated reduction rules. Consider, for example, the WF-net shown in Fig. 10. As noted, the four silent transition to the left and right of the *A_FINALIZED+COMPLETE* transition can be merged into a single transition: If one of them fires, the other three should fire as well. If we merge them, then the existing reduction are capable of merging the four input places into a single input place and the four output places into a single output place, which makes the WF-net more simple and easier to understand. Another possible reduction would be to merge the two topmost silent transitions with the *A_DECLINE+COMPLETE* transition. Again, if one of them fires, then so should the others.

REFERENCES

[1] W. M. P. van der Aalst, A. J. M. M. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.

[2] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering Block-structured Process Models from Event Logs: A Constructive Approach," in *Applications and Theory of Petri Nets 2013*, J. Colom and J. Desel, Eds., vol. 7927, 2013, pp. 311–329.

[3] S. J. v. Zelst, B. F. v. Dongen, W. M. P. v. d. Aalst, and H. M. W. Verbeek, "Discovering workflow nets using integer linear programming," *Computing*, vol. 100, no. 5, pp. 529–556, May 2018. [Online]. Available: https://doi.org/10.1007/s00607-017-0582-5

[4] R. Bergenthum, "Prime miner - process discovery using prime event structures," in *2019 International Conference on Process Mining (ICPM)*, 2019, pp. 41–48.

[5] L. L. Mannel and W. M. P. van der Aalst, "Finding uniwired petri nets using est-miner," in *Business Process Management Workshops*, C. Di Francescomarino, R. Dijkman, and U. Zdun, Eds. Cham: Springer International Publishing, 2019, pp. 224–237.

[6] T. Hildebrandt and R. R. Mukkamala, "Declarative event-based workflow as distributed dynamic condition response graphs," *PLACES*, vol. 69, 10 2011.

[7] J. Desel and J. Esparza, *Free Choice Petri Nets*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1995, vol. 40.

[8] A. Augusto, J. Carmona, and H. M. W. Verbeek, "Advanced process mining," in *Process Mining Summerschool 2022*, ser. Lecture Notes in Business Information Processing (LNBIP), W. M. P. v. d. Aalst and J. Carmona, Eds. Aachen, Germany: Springer, July 4-8 2022, p. (to appear).

[9] W. M. P. van der Aalst, "The application of Petri nets to workflow management," *The Journal of Circuits, Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[10] J. L. Peterson, *Petri net theory and the modeling of systems*. Prentice-Hall, Englewood Cliffs, 1981.

[11] W. Reisig and G. Rozenberg, Eds., *Lectures on Petri Nets I: Basic Models*, vol. 1491, 1998.

[12] ——, *Lectures on Petri Nets II: Applications*, vol. 1492, 1998.

[13] W. M. P. van der Aalst, *Process Mining: Data Science in Action*, 2016.

[14] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, April 1989.

[15] H. M. W. Verbeek, "Decomposed replay using hiding and reduction as abstraction," *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, vol. XII, pp. 166–186, 2017. [Online]. Available: http://www.springerlink.com/content/f15t41545m061682/fulltext.pdf

[16] B. van Dongen. (2012, 4) BPI Challenge 2012. https://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2012/12689204

[17] J. J. Carmona, M. de Leoni, B. Depaire, and T. Jouck. (2021, 5) Process Discovery Contest 2016. https://dx.doi.org/10.4121/14625912.v1. [Online]. Available: https://data.4tu.nl/articles/dataset/Process_Discovery_Contest_2016/14625912

[18] ——. (2021, 5) Process Discovery Contest 2017. https://dx.doi.org/10.4121/14625948.v1. [Online]. Available: https://data.4tu.nl/articles/dataset/Process_Discovery_Contest_2017/14625948

[19] J. J. Carmona, M. de Leoni, and B. Depaire. (2021, 5) Process Discovery Contest 2019. https://dx.doi.org/10.4121/14625996.v1. [Online]. Available: https://data.4tu.nl/articles/dataset/Process_Discovery_Contest_2019/14625996

[20] H. M. W. Verbeek. (2021, 5) Process Discovery Contest 2020. https://dx.doi.org/10.4121/14626020.v1. [Online]. Available: https://data.4tu.nl/articles/dataset/Process_Discovery_Contest_2020/14626020

[21] ——. (2021, 10) Process Discovery Contest 2021. https://dx.doi.org/10.4121/16803232.v1. [Online]. Available: https://data.4tu.nl/articles/dataset/Process_Discovery_Contest_2021/16803232