Process Discovery for Structured Program Synthesis

Dell Zhang[*] **Alexander Kuhnle, Julian Richardson, Murat Sensoy**
dell.z@ieee.org
{alexander.kuhnle,julian.richardson,murat.sensoy}@blueprism.com

Blue Prism AI Labs

Abstract

A core task in process mining is process discovery which aims to learn an accurate process model from event log data. In this paper, we propose to use (block-) structured programs directly as target process models so as to establish connections to the field of program synthesis and facilitate the translation from abstract process models to executable processes, e.g., for robotic process automation. Furthermore, we develop a novel bottom-up agglomerative approach to the discovery of such structured program process models. In comparison with the popular top-down recursive inductive miner, our proposed agglomerative miner enjoys the similar theoretical guarantee to produce sound process models (without deadlocks and other anomalies) while exhibiting some advantages like avoiding silent activities and accommodating duplicate activities. The proposed algorithm works by iteratively applying a few graph rewriting rules to the directly-follows-graph of activities. For real-world (sparse) directly-follows-graphs, the algorithm has quadratic computational complexity with respect to the number of distinct activities. To our knowledge, this is the first process discovery algorithm that is made for the purpose of program synthesis. Experiments on the BPI-Challenge 2020 dataset and the Karel programming dataset have demonstrated that our proposed algorithm can outperform the inductive miner not only according to the traditional process discovery metrics but also in terms of the effectiveness in finding out the true underlying structured program from a small number of its execution traces.

1 Introduction

In recent years, there has been a surge of interest in *process mining* [31] and *robotic process automation* [33]. While the former addresses the problem of analyzing and optimizing processes, the latter tries to automate those mundane and repetitive tasks with software agents. In the vision of *hyperautomation*, predicted by Gartner in 2020 as the No. 1 strategic technology trend, advanced technologies including machine learning, process mining and robotic process automation need to be combined and coordinated in order to reap the full benefit of a digital workforce.

One area where process mining could be utilized to help robotic process automation is *demo-to-process* [10, 34], i.e., learning an executable software process from human demonstrations or *user-interaction logs* [24]. This could relieve developers of the manual effort to build processes for robotic process automation. However, although process mining and robotic process automation are widely regarded as “a perfect match” [11], it is not straightforward to seamlessly integrate them together. There is actually a gap between the process models discovered by today’s process mining techniques and the process models ready to be deployed for robotic process automation. For example, Petri net, which is probably the most popular process model used in process discovery, is alien to

[*] Dell Zhang is the corresponding author of this paper. He is on leave from Birkbeck, University of London, and works full-time for Blue Prism AI Labs.

the mainstream robotic process automation systems such as Blue Prism, UiPath and Automation Anywhere. It may be because existing process discovery techniques were designed for the processes of high-level business activities where complex phenomena like concurrency often occur and need to be captured, but for robotic process automation, the processes are of low-level user-interaction activities that must be able to be carried out by computer software.

In fact, the processes in robotic process automation, at least of today, are essentially (block-) *structured programs* that consist of simple control flow constructs. Here "structured" has the same meaning as in the term "structured programming" coined by the computer scientist Edsger Dijkstra. The well-known structured program theorem (aka the BöhmâĂŞJacopini theorem) [4] tells us that any computable function can be represented using three control flow constructs — sequence, selection and iteration — as the only building blocks. Why don't we use structured programs directly as the target process models for process discovery? Thus the discovered process models, i.e., the structured programs, can be fed straightaway to robotic process automation systems without any friction. Driven by the above motivation, we have developed a new process discovery algorithm that learns structured programs directly from event logs. The typical demo-to-process scenario is that a relatively simple process model (i.e., a short structured program) needs to be inferred from only a small number of demonstrations, which is quite different from the traditional process discovery problem setting where a relatively complex process model needs to be inferred from a large number of traces in the event log.

The existing process discovery technique most similar to what we propose in this paper is the *inductive miner* [19–22] which has *process trees* as its target process models. Although structured programs, or equivalently their *abstract syntax trees*, have the same expressive power as process trees, there are several nontrivial differences between them which make structured programs easier to understand and implement. Moreover, while the inductive miner recursively splits the *directly-follows-graph* [19, 23] — a graph that indicates what activities occurred right after what activities in the given event log — in a *top-down* fashion, our proposed process discovery algorithm works the other way around: it iteratively "condenses" the directly-follows-graph of activities *bottom-up*. That is why we name this approach *agglomerative process discovery*. In theory, there should be no fundamental difference between the hierarchical process models constructed top-down or bottom-up. However, in practice, we have found that the bottom-up approach is likely to generate better hierarchical process models than the top-down approach, probably because it is a lot easier to recognize local control flow constructs than global control flow constructs from the directly-follows-graph, as we will explain later.

For any input event log, our proposed process discovery algorithm finally outputs a program. Therefore, it can also be considered as a method for *program synthesis* [16], or more specifically, *programming by demonstration* [8, 25]. Following the steps of some recent work in neural program synthesis [6, 7, 9, 29], we use Karel, a simple educational programming language [27], as the testbed to evaluate our proposed agglomerative miner and compare it with the inductive miner for the purpose of structured program synthesis. The experimental results on large-scale public datasets are encouraging.

2 Related Work

2.1 Process Discovery

One of the most important and most studied problems in process mining [31] is process discovery, which tries to find a suitable process model to describe the control flow relations between the activities observed in or implied by a given event log [2]. It is straightforward, but not really useful, to produce a process model that matches only the observed traces in the given event log (i.e., 100% precision) or a process model that matches every possible trace (i.e., 100% recall). The central challenge for process discovery is to make the right trade-off and strike the optimal balance between *precision*, *recall* (more commonly known as *fitness* in the process mining literature), *generalization* and *simplicity* [31].

A well-known classic process discovery algorithm is the *alpha miner* (the α algorithm) [32]. It is able to find a Petri net model to fit the event log where all the activities are visible and unique. One notable weakness of the alpha miner and many other process discovery algorithms is that the discovered model may not be sound, i.e., the model could suffer from anomalies like deadlocks.

The most popular process discovery algorithm today is probably the *inductive miner* [19–21], especially its latest version based on directly-follows-graphs called IMD [22]. It produces a (block-)

structured hierarchical process tree as the output model. All the process trees are guaranteed to be sound, which might be the biggest strength of the inductive miner. The basic idea of the inductive miner is to recursively detect an appropriate "cut" to split the directly-follows-graph [19,23] top-down until the graph is divided into just individual activities (base cases). There are four possible types of cuts: sequence, exclusive-choice, redo-loop and parallel. Often the process tree has to introduce silent/hidden activities (τ) to capture the control flow, and it prohibits the existence of any duplicate activity.

2.2 Program Synthesis

The task of *program synthesis* [16] is to automatically construct a program (in the underlying programming language) that can satisfy a user intent expressed in some form of high-level specification. This sub-field of AI has a long history and it has been considered as the "holy grail" of computer science. In recent years, it has attracted a lot of attention due to the popularization of practical program synthesis applications (like the FlashFill feature in Microsoft Excel [14, 15]) and also the great potential of deep learning for neural program synthesis [6, 9]. Popular program synthesis frameworks include PROSE, SKETCH, ROSETTE and FOOFAH.

Most of the recent work in this area aims to learn simple programs (e.g., in the Karel programming language [27]) only from input-output examples or, in a couple of recent studies [7,29], by additionally exploiting the (inferred) execution traces to improve program synthesis. In process mining, we usually instead assume the availability of traces (e.g., stored in an event log) but not input-output examples.

3 Proposed Approach

3.1 Structured Program

The target process model for our proposed approach to process discovery is just (block-) structured programs that are formally defined in Table 1. The alphabet Σ is the finite set of activities that can occur in the event log, and the sole non-terminal symbol S represents a structured program. As shown by the production rules, S can be either a simple statement which consists of a single activity (terminal symbol), or a compound statement that is made from a control flow construct with smaller program pieces as its components.

To denote the three standard control flow constructs in structured programming, i.e., sequence, selection and iteration, we borrow the widely used *regular expression* operators. Specifically, the $?$ operator indicates an optional occurrence of its preceding statement S; the $|$ operator indicates the occurrence of either the statement on its left S_1 or the statement on its right S_2; the $+$ operator indicates one or more occurrences of its preceding statement S; and the $*$ operator indicates zero or more occurrences of its preceding statement S. Parentheses are used to group statements for the application of operators.

In addition to the above standard control flow constructs, we also include the concurrence construct (with the $\&$ operator) in order to represent the parallel execution of statements. This is necessary to make our process model comparable with the other process models in process discovery. Some programming languages like Erlang have built-in primitives to support concurrent or parallel computing. For the programming languages without this capability, such as Python, it can be implemented via an add-on library (e.g., `multiprocessing`) or simply backing off to the serialized execution of statements.

Such a structured program can be represented equivalently by its *abstract syntax tree*, where the leaf nodes are activities (simple statements) and the internal nodes are operators (to construct compound statements), as illustrated in Fig. 1.

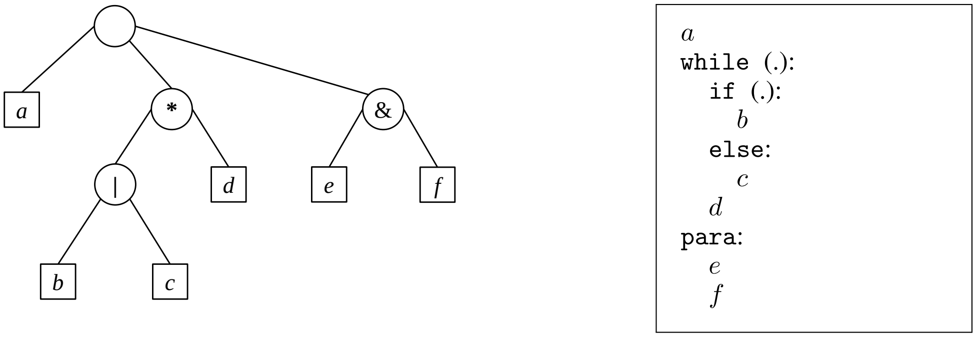
The (structured program) syntax tree model looks very similar to the *process tree* model used by the inductive miner [19]. Indeed, they should have the same expressive power, and they are both block-structured process models that are *sound* by construction. However, syntax trees are tailored towards program synthesis and differ from process trees in two important aspects. First, syntax trees completely avoid the usage of invisible *silent activities* (τ) which are unintuitive. Second, syntax trees describe iterations with the standard concept of while-loops (as in almost all programming

Table 1: The constituency grammar of structured programs.

Production Rule	Control Flow	Source Code
$S \longrightarrow x$	activity $x \in \Sigma$	x
$S \longrightarrow (S_1 S_2)$	sequence	S_1 S_2
$S \longrightarrow (S_1?)$	selection	if (.): S_1

$S \longrightarrow (S_1 \vert S_2)$	selection	if (.):
		  S_1
		else:
		  S_2

$S \longrightarrow (S_1+)$	iteration	while (.): S_1
$S \longrightarrow (S_1*)$	iteration	while (.): S_1
$S \longrightarrow (S_1 \& S_2)$	concurrence	para: S_1 S_2



```
a
while (.):
  if (.):
    b
  else:
    c
  d
para:
  e
  f
```

(a) abstract syntax tree

(b) source code

Figure 1: An example structured program $(a((b|c)d)*(e\&f))$.

languages), instead of the obscure *redo-loops* which consists of not only a “do” part but also one or more “re-do” parts. Therefore, syntax trees are easier to interpret and implement.

To a large degree, the (structured program) syntax tree model also resembles *regular expressions* (defined by a *regular language*), except that syntax trees can also model concurrency with the additional & operator. A well-known theorem in computer science established by E Mark Gold states that even regular expressions cannot be *learned in the limit* from positive examples only [12], though the problem of *inductive inference* has been investigated for a variety of subclasses [1]. Most existing process discovery algorithms seem to overcome this obstacle to learn from positive examples (observed traces) only by imposing a strong *inductive bias* against duplicate activities in the process model.

3.2 Agglomerative Miner

Our proposed agglomerative approach to process discovery is given in Algorithm 1. Let us explain it in detail and compare it with the inductive miner [22].

As with all existing process discovery algorithms, the agglomerative miner takes an event log L as the input and produces a process model S as the output. Here, the input event log L is a bag (multiset) of *traces*, each of which consists of a sequence of activities, and the output process model S will be a structured program (or equivalently its abstract syntax tree).

```
Algorithm 1: Agglomerative Process Discovery
Input  : An event log $L$.
Output : A structured program $S$.
for each trace $\langle a_1, \ldots, a_k \rangle \in L$ do
    Expand it to $\langle \wedge, a_1, \ldots, a_k, \$ \rangle$ where $\wedge$ and $ are the special 'begin' and 'end' activities
      respectively;
end
Construct the directly-follows-graph $G$ for $S$;
while not converged do
    repeat
        Condense $G$ using the graph rewriting rule Fig. 3a iteration1 (self-loop);
    until $G$ cannot be condensed further;
    repeat
        Condense $G$ using the graph rewriting rule Fig. 3b sequence;
    until $G$ cannot be condensed further;
    repeat
        Condense $G$ using the graph rewriting rules Figs. 3c to 3g iteration2-6 (general-loop) as well
          as Fig. 3h concurrence;
    until $G$ cannot be condensed further;
    repeat
        Condense $G$ using the graph rewriting rule Fig. 3i: selection1 (multi-branch);
    until $G$ cannot be condensed further;
    repeat
        Condense $G$ using the graph rewriting rule Fig. 3j: selection2 (single-branch);
    until $G$ cannot be condensed further;
end
if $G$ contains more than one node other than $\wedge$ and $ then
    Condense $G$ using the fall-through "flower" model as the last resort;
end
$S \longleftarrow$ the structured program saved at the node $v$, the only node left other than $\wedge$ and $;
return $S$
```

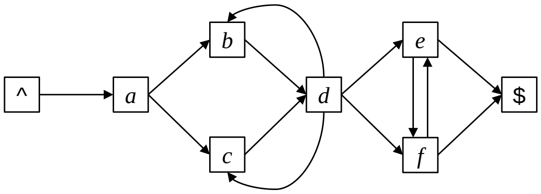
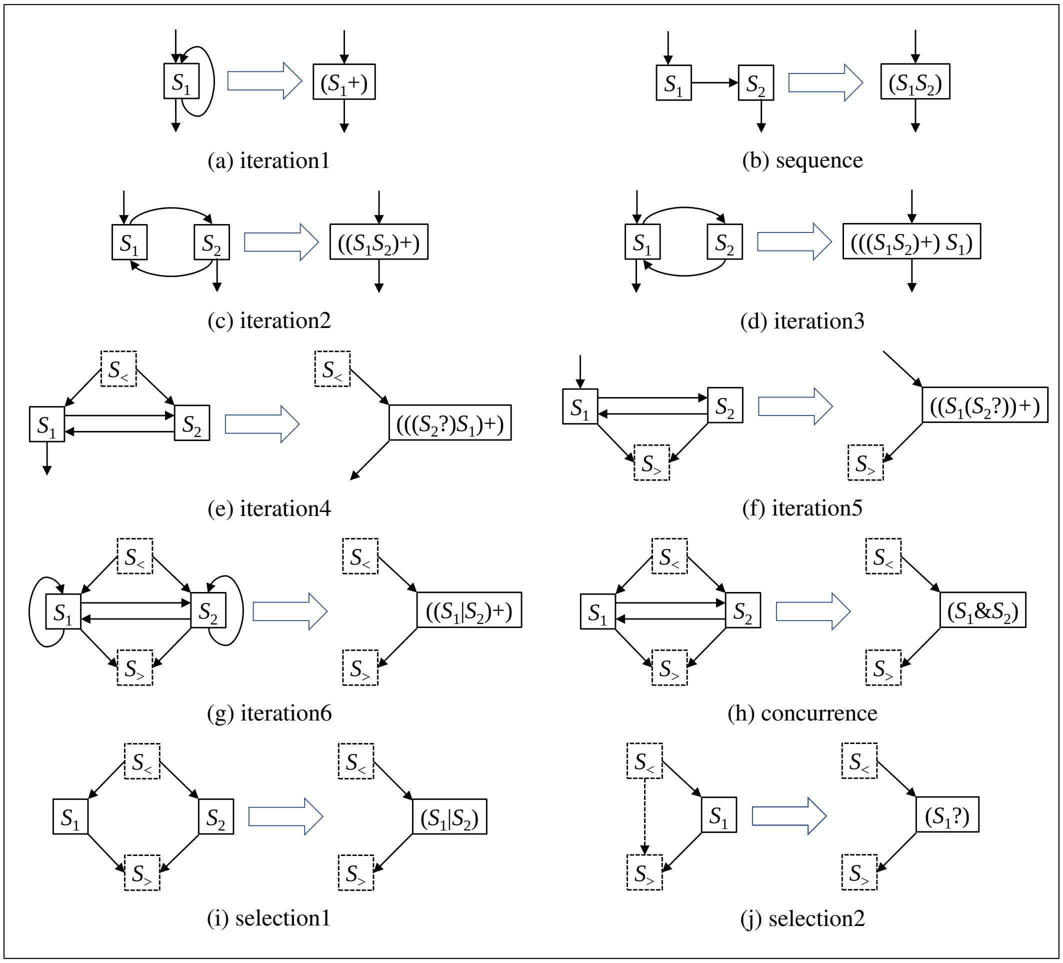


Figure 2: The directly-follows-graph constructed from the execution traces of the example structured program.

Similar to the inductive miner, the agglomerative miner first converts the given event log into a *directly-follows-graph* which has a directed edge (link) from a node (activity) u to another node (activity) v, if and only if, u is directly followed by v. Unlike the inductive miner, the agglomerative miner does not need to memorize the start and end activities of the directly-follows-graph. Instead, we introduce two special activities $\wedge$ and $ to represent the beginning and end of traces respectively, which simplifies the algorithm. Fig. 2 shows the directly-follows-graph corresponding to the example structured program in Fig. 1.

The body of the agglomerative process discovery algorithm is an iterative procedure of graph rewriting that condenses the directly-follows-graph step by step until only one node (other than $\wedge$ and $) is left. Along with this iterative procedure, the activities represented by the nodes of the directly-follows-graph are pieced together through different control flow constructs gradually into a complete structured program, which is the final inferred process model. The overall framework in which the input graph is summarized into a single node containing the output bears some similarity to the *state*



(a) iteration1

(b) sequence

(c) iteration2

(d) iteration3

(e) iteration4

(f) iteration5

(g) iteration6

(h) concurrence

(i) selection1

(j) selection2

Figure 3: The graph rewriting rules for the agglomerative process discovery algorithm.

elimination method for transforming nondeterministic finite automata into regular expressions [13,17] that can be traced back to Kleene [18].

Fig. 3 shows all the graph rewriting rules employed by the agglomerative miner. Each of them detects a particular local graph pattern and then uses the corresponding control flow construct to condense the graph. For example, as shown in Fig. 3b, if we see that a node S_1 has only one successor S_2 and the node S_2 has only one predecessor S_1, we can safely merge S_1 and S_2 into a sequence (S_1S_2). Those graph rewriting rules cover all the basic local graph patterns involving either one or two nodes. The order in which they are applied matters: the general principle here is that the graph rewriting rules with less ambiguity should be applied before those with more ambiguity. Since there is no uncertainty about the self-loop pattern Fig. 3a and the sequence pattern Fig. 3b, these are always attempted first at each iteration of the algorithm.

One particular graph rewriting rule, “iteration3” (Fig. 3d), is of particular importance, as it leads to a piece of structured program $(((S_1S_2)+)S_1)$ which contains the duplication of statement S_1. This distinguishes the agglomerative miner from most existing process discovery algorithms including the inductive miner, as they do not accommodate *duplicate activities* in the process model. When S_1 is a simple statement (containing just an activity), duplicating S_1 is preferable because it results in a relatively small process model that can faithfully fit this local graph pattern. However, when S_1 is a compound statement (containing multiple activities), duplicating S_1 would make the generated process model much bigger which is probably undesirable, so we may actually want to sacrifice precision for simplicity and resolve to $((S_1(S_2?))+)$ instead.

Table 2: The simplification of structured programs.

Original Expression	⇒	Simplified Expression
$((S?)?)$	$\Rightarrow$	$(S?)$
$((S+)?)$	$\Rightarrow$	$(S*)$
$((S*)?)$	$\Rightarrow$	$(S*)$
$((S+)+)$	$\Rightarrow$	$(S+)$
$((S?)+)$	$\Rightarrow$	$(S*)$
$((S*)+)$	$\Rightarrow$	$(S*)$
$((S?)*)$	$\Rightarrow$	$(S*)$
$((S+)*)$	$\Rightarrow$	$(S*)$
$((S*)*)$	$\Rightarrow$	$(S*)$
$((S_1S_2)S_3)$	$\Rightarrow$	$(S_1S_2S_3)$
$(S_1(S_2S_3))$	$\Rightarrow$	$(S_1S_2S_3)$
$((S_1\vert S_2)\vert S_3)$	$\Rightarrow$	$(S_1\vert S_2\vert S_3)$
$(S_1\vert (S_2\vert S_3))$	$\Rightarrow$	$(S_1\vert S_2\vert S_3)$
$((S_1\&S_2)\&S_3)$	$\Rightarrow$	$(S_1\&S_2\&S_3)$
$(S_1\&(S_2\&S_3))$	$\Rightarrow$	$(S_1\&S_2\&S_3)$
$((S_1?)\vert S_2)$	$\Rightarrow$	$((S_1\vert S_2)?)$
$(S_1\vert (S_2?))$	$\Rightarrow$	$((S_1\vert S_2)?)$
$(((S_1+)\vert S_2)+)$	$\Rightarrow$	$((S_1\vert S_2)+)$
$((S_1\vert (S_2+))+)$	$\Rightarrow$	$((S_1\vert S_2)+)$
$(((S_1*)\vert S_2)+)$	$\Rightarrow$	$((S_1\vert S_2)*)$
$((S_1\vert (S_2*))+)$	$\Rightarrow$	$((S_1\vert S_2)*)$
$(((S_1+)\vert S_2)*)$	$\Rightarrow$	$((S_1\vert S_2)*)$
$((S_1\vert (S_2+))*)$	$\Rightarrow$	$((S_1\vert S_2)*)$
$(((S_1*)\vert S_2)*)$	$\Rightarrow$	$((S_1\vert S_2)*)$
$((S_1\vert (S_2*))*)$	$\Rightarrow$	$((S_1\vert S_2)*)$
$(((S_1?)(S_2?))+)$	$\Rightarrow$	$((S_1\vert S_2)*)$
$(((S_1*)(S_2*))+)$	$\Rightarrow$	$((S_1\vert S_2)*)$
$(((S_1?)(S_2?))*)$	$\Rightarrow$	$((S_1\vert S_2)*)$
$(((S_1*)(S_2*))*)$	$\Rightarrow$	$((S_1\vert S_2)*)$
$(((S_1+)(S_2?))+)$	$\Rightarrow$	$((S_1(S_2?))+)$
$(((S_1?)(S_2+))+)$	$\Rightarrow$	$(((S_1?)S_2)+)$
$(((S_1+)(S_2?))*)$	$\Rightarrow$	$((S_1(S_2?))*)$
$(((S_1?)(S_2+))*)$	$\Rightarrow$	$(((S_1?)S_2)*)$

Each time we utilize a graph rewriting rule to condense the directly-follows-graph, we also try to simplify the piece of structured program to be produced. Table 2 lists the simplification rules, each of which converts the given structured program into a semantically equivalent but syntactically simpler one. For example, if the final operator used by a graph rewriting rule is '?', then the first three simplification rules would be applicable.

The above described iterative graph rewriting procedure is guaranteed to converge, as the application of each graph rewriting rule reduces the directly-follows-graph by either eliminating a node (i.e., contracting a pair of nodes into one) or eliminating an edge. Therefore the number of iterations is bounded by the size of the graph, and in practice a few iterations are usually enough to reach convergence.

In case there are still more than one node (other than $\wedge$ and \$) after the iterative graph rewriting procedure has converged, we summarize those remaining intermediate nodes $S_1, \ldots, S_k$ using the fall-through "flower" model $((S_1|\ldots|S_k)+)$. Since this so-called flower model can fit any trace of activities, it is used as the last resort by many process discovery algorithms including the inductive miner.

In some graph rewriting rules (Figs. 3e to 3j), we require the two nodes involved have a common predecessor ($S_<$) or/and a common successor ($S_>$), which is to ensure that the generated control flow construct has a well-defined entry and exit point. Moreover, for the graph rewriting rule Fig. 3i (selection with two branches), the above constraint also helps to reduce the computational complexity: without this constraint, detecting the corresponding local graph pattern would require us to enumerate all the possible node pairs; with this constraint, however, we only need to check the successors of each node, which is much cheaper computationally.

Overall, for an event log with n distinct activities, the computational complexity of the agglomerative process discovery algorithm is $O(n^2)$ if the directly-follows-graph is sparse (which is likely to be true for real-world datasets), or $O(n^3)$ otherwise. The reasoning is as follows. The construction of the directly-follows-graph can be easily done with a sequential scan of the event log, so its time cost is negligible. The number of nodes in the directly-follows-graph $|V|$ is obviously just the number of distinct activities n. The number of edges $|E|$ is at most n^2. For "sparse" graphs (as commonly defined in graph theory or network science), $|E|$ is at the level of $O(n)$ instead of $O(n^2)$. The most computationally expensive part of the algorithm, the graph rewriting procedure, has up to $|V| + |E|$ iterations, as explained earlier. It is not difficult to see that at each iteration, the application of all the graph rewriting rules requires at most $O(|V| + |E|)$ steps. Consequently, the total computational complexity is $O((|V| + |E|)^2) = O(n^2)$ for sparse directly-follows-graphs.

In summary, our proposed agglomerative miner is inspired by the popular inductive miner – especially its latest version based on directly-follows-graphs – but it is different from the inductive approach in a range of nontrivial aspects: it works bottom-up rather than top-down; it relies on iteration rather than recursion; it outputs syntax trees instead of process trees; and it avoids silent activities but accommodates duplicate activities in the final process model.

The agglomerative and inductive miner are both guaranteed to produce *sound* process models (without deadlocks and other anomalies). Why is it possible for the bottom-up agglomerative approach to find better process models than the top-down inductive approach? We conjecture that it is because the inductive miner often has to make hard choices among different possible control flow constructs at early stages. By contrast, the agglomerative miner starts from extracting the obvious (unambiguous) local graph patterns using fine-grained graph rewriting rules which simplify the graph, then in subsequent iterations previously complex (ambiguous) graph patterns become straightforward in the simplified graph and thus can be further collapsed; this iterative graph rewriting procedure continues until the entire directly-follows-graph is summarized into a single piece of structured program. While the inductive miner employs only four global graph patterns for recursive graph splitting, the agglomerative miner defines ten local graph patterns (as listed in Fig. 3), and more could be added if necessary.

4 Experiments

We have conducted experiments on two datasets, one in the traditional process discovery setting and the other for the purpose of program synthesis, to empirically evaluate the proposed agglomerative miner and compare it with existing process discovery methods, including the classic alpha miner and the popular inductive miner. For the inductive miner, we are referring to its latest version based on directly-follows-graphs, IMD [22], as it is scalable and most similar to our proposed agglomerative miner.

The agglomerative miner is implemented in Python 3, and we take implementations of the alpha miner as well as the inductive miner (specifically IMDFc) from the open-source Python library PM4Py[2].

4.1 BPI-Challenge 2020

The BPI-Challenge 2020 dataset[3] is a newly released public benchmark dataset for process mining. It contains five large-scale event logs pertaining to two years of travel expense claims at the Eindhoven University of Technology (TU/e).

We adopt the following standard performance metrics for automated process discovery which have been widely used in the process mining research literature [2]: fitness, precision [26], F_1-score, generalization [5], and simplicity [3]. Among them, F_1-score is the harmonic mean of fitness (i.e., recall) and precision which reflects the overall accuracy of process discovery.

Tables 3 and 4 show the dataset statistics and the experimental results of the three process mining algorithms in comparison. It can be clearly seen that the agglomerative miner achieves the best F_1-score as well as simplicity on all five event logs, and it is also the best-performing model with respect

[2] https://pm4py.fit.fraunhofer.de/

[3] https://icpmconference.org/2020/bpi-challenge/

Table 3: Descriptive statistics of the BPI-Challenge 2020 dataset.

Log	#Cases	#Events
Domestic Declarations	10,500	56,437
International Declarations	6,449	72,151
Prepaid Travel Cost	2,099	18,246
Request for Payment	6,886	36,796
Travel Permits	7,065	86,581

Table 4: Process discovery performances on the BPI-Challenge 2020 dataset.

Log	Method	Fitness	Precision	F_1-score	Generalization	Simplicity
Domestic Declarations	Alpha	0.7063	0.2500	0.3693	0.7851	0.1494
	Inductive	0.9400	0.3086	0.4647	0.7540	0.5769
	Agglomerative	1.0000	0.3286	**0.4946**	**0.8304**	**0.5814**
International Declarations	Alpha	0.5786	0.0000	0.0000	0.8987	0.1296
	Inductive	1.0000	0.0958	0.1748	**0.9029**	0.5273
	Agglomerative	0.8279	0.2550	**0.3899**	0.9004	**0.6048**
Prepaid Travel Cost	Alpha	0.6456	0.0000	0.0000	**0.8773**	0.0554
	Inductive	0.9982	0.1111	0.2000	0.8700	0.5238
	Agglomerative	0.8294	0.1736	**0.2871**	0.8675	**0.5942**
Request for Payment	Alpha	0.7610	0.0000	0.0000	0.7836	0.1396
	Inductive	0.9317	0.2168	0.3517	0.8038	0.5842
	Agglomerative	0.9340	0.2284	**0.3670**	**0.8805**	**0.6577**
Travel Permits	Alpha	0.5850	0.0000	0.0000	0.8553	0.1725
	Inductive	0.9996	0.0708	0.1323	0.8110	0.4912
	Agglomerative	0.9811	0.1146	**0.2052**	**0.8952**	**0.5000**

$$\begin{aligned} \text{Prog } p &:= \texttt{def run}() : s \\ \text{Stmt } s &:= a \mid s_1; s_2 \mid \texttt{if } (b) : s \mid \texttt{if } (b) : s_1 \ \texttt{else} : s_2 \mid \texttt{while } (b) : s \mid \texttt{repeat } (r) : s \\ \text{Cond } b &:= \texttt{frontIsClear}() \mid \texttt{leftIsClear}() \mid \texttt{rightIsClear}() \mid \texttt{markersPresent}() \mid \\ & \ \texttt{noMarkersPresent}() \mid \texttt{not } b \\ \text{Action } a &:= \texttt{move}() \mid \texttt{turnRight}() \mid \texttt{turnLeft}() \mid \texttt{pickMarker}() \mid \texttt{putMarker}() \\ \text{Cste } r &:= 0 \mid 1 \mid \cdots \mid 19 \end{aligned}$$

Figure 4: The domain-specific language for Karel programs [6].

to generalization on three out of five event logs. This confirms the effectiveness of our proposed agglomerative miner for traditional process discovery with many traces and complex models.

4.2 Karel Programming

Karel, an educational programming language for beginners [27], has been utilized as the testbed by some recent research work in deep learning for neural program synthesis [6, 7, 9, 29]. It features a "robot" living in a grid-world who can move forward, turn left or right, and pick up or put down markers. The grammar of the Karel language is shown in Fig. 4. This is obviously a structured programming language with the control flow constructs sequence, selection and iteration. In this paper, we focus on discovering a Karel program's control flow structure from a small number of extraction traces, but leave the logical conditions (for selections or iterations) to future work (see Section 5). The `while` and `repeat` loops are both mapped to the iteration operators ($+$ or $*$) of the structured program process model as defined in Section 3.1.

Table 5: Descriptive statistics of the filtered Karel programming dataset (where each log has 6 traces).

Data Subset	#Logs	Trace-Length	Prog-Tokens	Tree-Depth
None-Duplicate	9,088	12.32±09.50	5.44±1.65	3.95±0.79
With-Duplicate	25,714	18.93±13.57	10.32±4.19	4.35±0.96

Table 6: Process discovery performances on the filtered Karel programming dataset.

Data Subset	Method	Fitness	Precision	F_1-score	Generalization	Simplicity
None-Duplicate	Inductive	0.9959	0.5338	0.6832	0.6101	0.8264
	Agglomerative	0.9903	0.7582	**0.8433**	**0.6854**	**0.8969**
With-Duplicate	Inductive	0.9888	0.4146	0.5717	0.6359	0.7837
	Agglomerative	0.9789	0.5209	**0.6498**	**0.6925**	**0.8027**

The Karel programming dataset[4] is a large dataset of simple Karel programs used for training and testing the models synthesizing Karel programs from input-output examples. We adapt this dataset for synthesizing Karel programs from execution traces instead. Each Karel program in the dataset comes with six input-output examples. For our experiments, we compute the six execution traces (i.e., the sequence of actions) for each Karel program with respect to those six input-output examples, and then filter out the Karel programs which have less than six distinct execution traces. Thus, we obtain a large set of event logs where each event log contains six traces (cases) generated by a ground-truth Karel program. Furthermore, we split the set of event logs into two subsets according to whether the corresponding ground-truth Karel program has duplicate activities or not. This is to facilitate the investigation of how important it is to accommodate duplicate activities in the process model. Table 5 shows the descriptive statistics of the *filtered* Karel programming dataset. For each subset, we have calculated the average length of execution traces, the average number of ground-truth program tokens and the average depth of ground-truth abstract syntax trees.

To the best of our knowledge, the inductive miner is the only existing process discovery algorithm that can produce structured programs. Therefore only the inductive miner is included as the baseline in our experiments on the Karel programming dataset.

As shown in Table 6, our proposed agglomerative miner significantly outperforms the inductive miner on both subsets in terms of the standard process discovery performance metrics F_1-score, generalization and simplicity.

More importantly, we propose to measure the performance of structured program synthesis by comparing the structured program (process model) generated by a process discovery algorithm with the ground-truth. One metric is the proportion of *exact matches*, i.e., what percentage of generated programs are exactly identical to the true underlying programs. Since the order of different branches in the selection control flow construct should not affect its semantics, i.e., $(S_1|S_2)$ is equivalent to $(S_2|S_1)$, we sort the branches in all the `if-then-else` statements beforehand to disregard such superficial differences. Another metric is the Levenshtein *edit distance* between each generated program and its corresponding ground-truth program. Here we consider each program as a sequence of program tokens rather than a string of characters, so an edit means an insertion, deletion or substitution of not a single character but a single program token. The smaller the edit distance, the better the generated program, as it is closer to the ground-truth. Note that both of the above two metrics measure the *syntactic* similarity/discrepancy between programs, which is an underestimation of the effectiveness for program synthesis: it is very possible for two syntacticly different programs to be semantically equivalent (known as *program aliasing* [6]). Nevertheless, these two syntactic metrics are obviously still informative and useful.

As shown in Table 7, our proposed agglomerative miner works significantly better than the inductive miner for the Karel program synthesis task in terms of both exact matches and edit distances. If the ground-truth program does not contain duplicate statements (activities), the agglomerative miner can recover it exactly from six traces with a good ($> 55\%$) chance, which is about ten times higher than the inductive miner baseline. Even when the ground-truth program contains duplicate statements

[4] https://msr-redmond.github.io/karel-dataset/

Table 7: Program synthesis performances on the filtered Karel programming dataset.

Data Subset	Method	Exact-Match	Edit-Dist
None-Duplicate	Inductive	517/9088 = 5.69%	2.94±1.80
	Agglomerative	5123/9088 = **56.37%**	**1.24**±1.73
With-Duplicate	Inductive	0/25714 = 0.00%	7.18±3.20
	Agglomerative	258/25714 = **1.00%**	**5.98**±3.61

(activities), the agglomerative miner still manages to get 1% exact matches, thanks to the graph rewriting rule Fig. 3d.

5 Future Work

The agglomerative process discovery algorithm needs to be extended to address the *infrequency* and *incompleteness* of behavior, i.e., the activities that are rarely observed and thus tend to be outliers as well as the activities that have not been recorded in the event log. In principle, similar techniques from the inductive miner [20–22] could be utilized.

This paper has focused on the discovery/synthesis of a program's control flow structure only, but ignored the inference of logical conditions for selection and iteration. It is possible to derive such logical conditions by analyzing the states of the environment at and before the point the process branches into different paths according to the recorded execution traces. For Karel programs, the state at any moment could be fully specified by four Boolean variables: `frontIsClear`, `leftIsClear`, `rightIsClear` and `markersPresent` (see Fig. 4). The *decision tree* learning algorithm is promising to address this problem, as shown by previous studies [28, 30].

When we evaluate process discovery algorithms for their effectiveness in program synthesis, we have only measured the syntactic equivalence between the generated program and the ground-truth program. Ideally, we want to measure the semantic equivalence: whether the two given programs would exhibit identical behavior, i.e., always produce the same output for the same input. This metric is partially reflected by the previously mentioned generalization score for process discovery, but a more accurate way to estimate it is to execute two given programs under a large number of conditions and compare their outputs.

The program synthesis experimental results in Section 4.2 suggest that duplicate statements (activities) are common in real-world structured programs (process models) but they are not well addressed by existing process discovery algorithms or the current version of agglomerative miner. This seems to be an important and challenging research problem in the direction towards a unified theory of process discovery and program synthesis.

6 Conclusion

The main contributions of this paper are as follows.

- First, we re-examine process discovery from the perspective of program synthesis, and argue that using structured programs directly as target process models would make the translation from abstract process models to executable processes easier to understand and implement, particularly in the context of robotic process automation.

- Second, we design an agglomerative process discovery algorithm for structured programs based on iterative graph rewriting, inspired by the popular inductive miner.

- Third, we introduce an evaluation framework for measuring the program synthesis performance of different process discovery algorithms, and demonstrate the advantages of our proposed agglomerative approach over existing methods.

References

[1] D. Angluin and C. H. Smith. Inductive Inference: Theory and Methods. *ACM Computing Surveys (CSUR)*, 15(3):237–269, 1983.

[2] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, F. M. Maggi, A. Marrella, M. Mecella, and A. Soo. Automated Discovery of Process Models from Event Logs: Review and Benchmark. *IEEE Transactions on Knowledge and Data Engineering*, 31(4):686–705, 2018.

[3] F. Blum. Metrics in Process Discovery. Technical report, Tech. Rep. TR/DCC. 1–21, 2015.

[4] C. Böhm and G. Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Communications of the ACM*, 9(5):366–371, 1966.

[5] J. C. Buijs, B. F. van Dongen, and W. M. van der Aalst. Quality Dimensions in Process Discovery: The Importance of Fitness, Precision, Generalization and Simplicity. *International Journal of Cooperative Information Systems*, 23(01):1440001, 2014.

[6] R. Bunel, M. Hausknecht, J. Devlin, R. Singh, and P. Kohli. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. *arXiv:1805.04276 [cs, stat]*, May 2018.

[7] X. Chen, C. Liu, and D. Song. Execution-Guided Neural Program Synthesis. In *International Conference on Learning Representations*, Sept. 2018.

[8] A. Cypher and D. C. Halbert. *Watch What I Do: Programming by Demonstration*. MIT press, 1993.

[9] J. Devlin, R. R. Bunel, R. Singh, M. Hausknecht, and P. Kohli. Neural Program Meta-Induction. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 2080–2088. Curran Associates, Inc., 2017.

[10] D. Ferreira, J. Rozanova, K. Dubba, D. Zhang, and A. Freitas. On the Evaluation of Intelligent Process Automation. *arXiv:2001.02639 [cs]*, Jan. 2020.

[11] J. Geyer-Klingeberg, J. Nakladal, F. Baldauf, F. Veit, W. M. P. van der Aalst, F. Casati, R. Conforti, M. de Leoni, and M. Dumas. Process Mining and Robotic Process Automation: A Perfect Match. In *BPM (Dissertation/Demos/Industry)*, pages 124–131, 2018.

[12] E. M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.

[13] J. L. Gross, J. Yellen, and P. Zhang. *Handbook of Graph Theory*. CRC press, 2013.

[14] S. Gulwani. Automating String Processing in Spreadsheets Using Input-Output Examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[15] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet Data Manipulation Using Examples. *Communications of the ACM*, 55(8):97–105, 2012.

[16] S. Gulwani, O. Polozov, and R. Singh. Program Synthesis. *Foundations and Trends in Programming Languages*, 2017.

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson International Edition. Addison-Wesley, 3rd edition, 2007.

[18] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies, Annals of Math. Studies 34*, pages 37–40, 1956.

[19] S. J. Leemans, D. Fahland, and W. M. van der Aalst. Discovering Block-Structured Process Models from Event Logs — A Constructive Approach. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 311–329. Springer, 2013.

[20] S. J. Leemans, D. Fahland, and W. M. van der Aalst. Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour. In *International Conference on Business Process Management*, pages 66–78. Springer, 2013.

[21] S. J. Leemans, D. Fahland, and W. M. van der Aalst. Discovering Block-Structured Process Models from Incomplete Event Logs. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 91–110. Springer, 2014.

[22] S. J. Leemans, D. Fahland, and W. M. van der Aalst. Scalable Process Discovery with Guarantees. In *Enterprise, Business-Process and Information Systems Modeling*, pages 85–101. Springer, 2015.

[23] S. J. Leemans, E. Poppe, and M. T. Wynn. Directly Follows-based Process Mining: Exploration & A Case Study. In *2019 International Conference on Process Mining (ICPM)*, pages 25–32. IEEE, 2019.

[24] V. Leno, A. Polyvyanyy, M. Dumas, M. La Rosa, and F. M. Maggi. Robotic Process Mining: Vision and Challenges. *Business & Information Systems Engineering*, Mar. 2020.

[25] H. Lieberman. *Your Wish Is My Command: Programming By Example*. Morgan Kaufmann, San Francisco, 1st edition edition, Mar. 2001.

[26] J. Muñoz-Gama and J. Carmona. A Fresh Look at Precision in Process Conformance. In *International Conference on Business Process Management*, pages 211–226. Springer, 2010.

[27] R. E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., 1981.

[28] A. Rozinat and W. M. van der Aalst. Decision Mining in ProM. In *International Conference on Business Process Management*, pages 420–425. Springer, 2006.

[29] R. Shin, I. Polosukhin, and D. Song. Improving Neural Program Synthesis with Inferred Execution Traces. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 8917–8926. Curran Associates, Inc., 2018.

[30] R. Shraga, A. Gal, D. Schumacher, A. Senderovich, and M. Weidlich. Inductive Context-aware Process Discovery. In *2019 International Conference on Process Mining (ICPM)*, pages 33–40. IEEE, 2019.

[31] W. van der Aalst. *Process Mining: Data Science in Action*. Springer, New York, NY, 2nd ed edition, May 2016.

[32] W. Van der Aalst, T. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[33] W. M. P. van der Aalst, M. Bichler, and A. Heinzl. Robotic Process Automation. *Business & Information Systems Engineering*, 60(4):269–272, Aug. 2018.

[34] D. Zhang, A. Freitas, D. Tao, and D. Song. Proceedings of the AAAI-20 Workshop on Intelligent Process Automation (IPA-20). *arXiv:2001.05214 [cs]*, Feb. 2020.