Recursion aware modeling and discovery for hierarchical software event log analysis (extended)

Leemans, M.; van der Aalst, W.M.P.; van den Brand, M.G.J.

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

Recursion Aware Modeling and Discovery
For Hierarchical Software Event Log Analysis (Ext.)

Technical Report version with guarantee proofs for the discovery algorithms

Maikel Leemans
Eindhoven University of Technology
Eindhoven, The Netherlands
Email: m.leemans@tue.nl

Wil M. P. van der Aalst
Eindhoven University of Technology
Eindhoven, The Netherlands
Email: W.M.P.v.d.Aalst@tue.nl

Mark G. J. van den Brand
Eindhoven University of Technology
Eindhoven, The Netherlands
Email: M.G.J.v.d.Brand@tue.nl

Abstract—This extended paper presents 1) a novel hierarchy and recursion extension to the process tree model; and 2) the first, recursion aware process model discovery technique that leverages hierarchical information in event logs, typically available for software systems. This technique allows us to analyze the operational processes of software systems under real-life conditions at multiple levels of granularity. The work can be positioned in-between reverse engineering and process mining. An implementation of the proposed approach is available as a ProM plugin. Experimental results based on real-life (software) event logs demonstrate the feasibility and usefulness of the approach and show the huge potential to speed up discovery by exploiting the available hierarchy.

Keywords-Reverse Engineering; Process Mining; Recursion Aware Discovery; Event Log; Hierarchical Event Log; Process Discovery; Hierarchical Discovery; Hierarchical Modeling

I. INTRODUCTION

System comprehension, analysis, maintenance, and evolution are largely dependent on information regarding the structure, behavior, operation, and usage of software systems. To understand the operation and usage of a system, one has to observe and study the system "on the run", in its natural, real-life production environment. To understand and maintain the (legacy) behavior when design and documentation are missing or outdated, one can observe and study the system in a controlled environment using, for example, testing techniques. In both cases, advanced algorithms and tools are needed to support a model driven reverse engineering and analysis of the behavior, operation, and usage. Such tools should be able to support the analysis of performance (timing), frequency (usage), conformance and reliability in the context of a behavioral backbone model that is expressive, precise and fits the actual system. This way, one obtains a reliable and accurate understanding of the behavior, operation, and usage of the system, both at a high-level and a fine-grained level.

The above design criteria make *process mining* a good candidate for the analysis of the actual *software behavior*. Process mining techniques provide a powerful and mature way to discover formal process models and analyze and improve these processes based on *event log* data from the system [46]. Event logs show the actual behavior of the system, and could be obtained in various ways, like, for example, instrumentation techniques. Numerous state of the art process mining techniques are readily available and can be used and combined through the Process Mining Toolkit ProM [53]. In addition, event logs are backed by the IEEE XES standard [22], [53].

Typically, the run-time behavior of a system is large and complex. Current techniques usually produce flat models that are not expressive enough to master this complexity and are often difficult to understand. Especially in the case of software systems, there is often a hierarchical, possibly recursive, structure implicitly reflected by the behavior and event logs. This hierarchical structure can be made explicit and should be used to aid model discovery and further analysis.

In this paper, *we 1) propose a novel hierarchy and recursion extension to the process tree model; and 2) define the first, recursion aware process model discovery technique that leverages hierarchical information in event logs, typically available for software systems*. This technique allows us to analyze the operational processes of software systems under real-life conditions at multiple levels of granularity. In addition, *the proposed technique has a huge potential to speed up discovery by exploiting the available hierarchy*. An implementation of the proposed algorithms is made available via the *Statechart* plugin for ProM [30]. The Statechart workbench provides an intuitive way to discover, explore and analyze hierarchical behavior, integrates with existing ProM plugins and links back to the source code in Eclipse.

This paper is organized as follows (see Figure 1). Section II positions the work in existing literature. Section III presents formal definitions of the input (event logs) and the proposed novel hierarchical process trees. In Section IV, we discuss how to obtain an explicit hierarchical structure. Two proposed novel, hierarchical process model discovery techniques are explained in Section V. In Section VI, we show how to filter, annotate, and visualize our hierarchical process trees. The approach is evaluated in Section VII using experiments and a small demo. Section VIII concludes the paper.
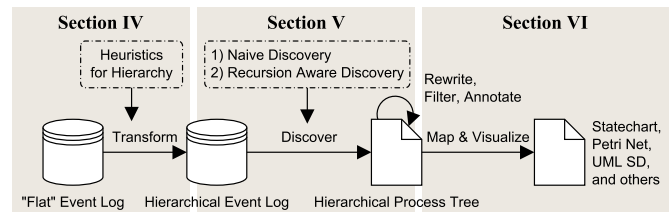

Fig. 1. Outline of the paper and our discovery approach.

TABLE I

COMPARISON OF RELATED TECHNIQUES AND THE EXPRESSIVENESS OF THE RESULTING MODELS, DIVIDED INTO THE GROUPS FROM SECTION II.

Group	Author	Technique / Toolkit	Input	Formalism	Execution Semantics	Model Quality	Aggregate Runs	Frequency Info	Performance Info	Choice	Loop	Concurrency	Hierarchy	Named Submodels	Recursion
Static Analysis	[45] Tonella	Object flow analysis	C++ source code	UML Interact.	-	n/a	n/a	-	-	-	-	-	-	-	-
	[27] Kollmann	Java code structures	Java source code	UML Collab.	-	n/a	n/a	-	-	-	-	-	-	-	-
	[28] Korshunova	CPP2XMI, SQuADT	C++ source code	UML SD, AD	-	n/a	n/a	-	-	✓	✓	-	-	-	-
	[40] Rountev	Dataflow analysis	Java source code	UML SD	±[1]	n/a	n/a	-	-	✓	✓	-	-	-	-
	[4] Amighi	Sawja framework	Java byte code	CFG	-	n/a	n/a	-	-	✓	✓	-	-	-	-
Dynamic Analysis	[3] Alalfi	PHP2XMI	Instrumentation	UML SD	±[1]	-	-	-	-	-	-	-	-	-	-
	[39] Oechsle	JAVAVIS	Java debug interface	UML SD	±[1]	-	-	-	-	-	-	✓	-	-	-
	[11] Briand	Meta models / OCL	Instrumentation	UML SD	±[1]	-	-	-	-	✓	✓	-	-	-	-
	[10] Briand	Meta models / OCL	Instrumentation	UML SD	±[1]	-	-	-	-	✓	✓	✓	-	-	-
	[29] Labiche	Meta models / OCL	Instrumentation + source	UML SD	±[1]	-	-	-	-	✓	✓	-	-	-	-
	[44] Systä	Shimba	Customized debugger	SD variant	±[1]	-	-	-	-	✓	✓	-	-	-	-
	[55] Walkinshaw	MINT	Log with traces	EFSM	✓	✓	✓	-	-	✓	✓	-	-	-	-
	[8] Beschastnikh	CSight	Given log, instrument	CFSM	✓	✓	✓	-	-	✓	✓	✓	✓	-	-
	[1] Ackermann	Behavior extraction	Monitor network packets	UML SD	±[1]	-	✓	-	✓	✓	✓	-	-	-	-
	[20] Graham	gprof profiler	Instrumentation	Call graphs	-	-	-	-	✓	-	✓	-	✓	✓	±[8]
	[17] De Pauw	Execution patterns	Program Trace	Exec. Pattern	-	-	-	✓	-	-	-	✓	✓	✓	✓
	[7] Beschastnikh	Synoptic	Log with traces	FSM	✓	✓	✓	✓	-	✓	✓	-	-	-	-
	[23] Heule	DFAsat	Log with traces	DFA	✓	✓	✓	-	-	✓	✓	-	-	-	-
Grammar	[38] Nevill-Manning	Sequitur	Symbol sequence	Grammar	✓	-	-	-	-	-	-	-	✓	-	-
	[41] Siyari	Lexis	Symbol sequence	Lexis-DAG	✓	-	-	✓	-	-	-	-	✓	-	-
	[26] Jonyer	SubdueGL	Symbol Graph	Graph Grammar	✓	-	-	-	-	✓	-	-	✓	-	±[9]
Process Mining	[49] Van der Aalst	Alpha algorithm	Event Log	Petri net	✓	-	✓	✓[2]	✓[2]	✓	✓	✓	-	-	-
	[48] Van der Aalst	Theory of Regions	Event Log	Petri net	✓	-	✓	✓[2]	✓[2]	✓	✓	✓	-	-	-
	[56] Weijters	Flexible heuristics miner	Event Log	Heuristics net	✓	-	✓	✓[2]	✓[2]	✓	✓	✓	-	-	-
	[50] Werf, van der	ILP miner	Event Log	Petri net	✓	✓	✓	✓[2]	✓[2]	✓	✓	✓	-	-	-
	[52] Zelst, S. J. van	ILP with filtering	Event Log	Petri net	✓	✓	✓	✓[2]	✓[2]	✓	✓	✓	-	-	-
	[47] Alves de Medeiros	Genetic Miner	Event Log	Petri net	✓	✓	✓	✓[2]	✓[2]	✓	✓	✓	-	-	-
	[12] Buijs	ETM algorithm	Event Log	Process tree	✓	✓	✓	✓[2]	✓[2]	✓	✓	✓	-	-	-
	[35] Leemans S.J.J.	Inductive Miner	Event Log	Process tree	✓	✓	✓	✓[2]	✓[2]	✓	✓	✓	-	-	-
	[21] Günther	Fuzzy Miner	Event Log	Fuzzy model	-	-	✓	✓[3]	-	✓	-	✓	✓[4]	-	-
	[25] Bose	Two-phase discovery	Event Log	Fuzzy model	-	-	✓	✓[3]	-	✓	-	✓	✓[5]	-	-
	[15] Conforti	BPMN miner	Event Log	BPMN	✓	✓	✓	±[2]	±[2]	✓	✓	✓	✓[6]	-	-
	This paper — Recursion Aware Disc.		Event Log	H. Process tree*	✓	✓	✓	✓[2]	✓[2]	✓	✓	✓	✓[7]	✓	✓

[1] Formal semantics are available for UML SD variants.
[2] Aligning an event log and a process model enables advanced performance, frequency, and conformance analysis, as described in [2], [35].
[3] Various log-based process metrics have been defined which capture different notions of frequency, significance, and correlation [21].
[4] The hierarchy is based on anonymous clusters in the resulting model [21].
[5] The hierarchy is based on abstraction patterns over events [24], [25].
[6] The hierarchy is based on discovered relations over extra data in the event log.
[7] The hierarchy is based on the hierarchical information in the event log.
[8] Recursion is detectable as a cycle, but without performance analysis support.
[9] Only tail recursion is supported.
* *Hierarchical process tree*, as introduced in Definition III.3.

II. RELATED WORK

Substantial work has been done on constructing models from software or example behavior in a variety of research domains. This section presents a brief comparison of various approaches and focuses mainly on the design criteria from the introduction. That is, the approach and tools should provide a behavioral backbone model that is expressive, precise and fits the actual system, and ideally should be able to support at least performance (timing) and frequency (usage) analysis. Table I summarizes the comparison.

A. Groups and Criteria for Comparison

We have divided the related work into four groups. *Static Analysis* utilizes static artifacts like source code files. *Dynamic Analysis* utilizes runtime information through instrumentation or tracing interfaces like debuggers. *Grammar Inference* relies on example behavior in the form of abstract symbol sequences. *Process Mining* relies on events logs and is in that sense a more implementation or platform agnostic approach.

For the comparison on design criteria, we define three sets of features. Firstly, a precise and fit model should: a) have *formal execution semantics*, and b) the underlying discovery algorithm should either guarantee or allow the user to control the *model quality*. The quality of a model is typically measured in terms of metrics like *fitness* and *precision*, but other qualities (e.g., simplicity and generalization) can also be considered [46]. Fitness expresses the part of the log (c.q. system behavior) that is represented by the model; precision expresses the behavior in the model that is present in the log (c.q. system behavior). Secondly, the model should be used as the backbone for further analysis. At the very least, *frequency* (usage) and *performance* (timing) analysis should be supported. In addition, the analysis should be (statistically) significant, and hence the technique should be able to *aggregate* information over multiple execution *runs*. Thirdly, the

model should be expressive and be able to capture the type of behavior encountered in software system cases. Not only should branching behavior like *choices* (e.g., if-then-else) and *loops* (e.g., foreach, iterators) be supported, but also hierarchy and recursion. Furthermore, for hierarchies, meaningful *names* for the different *submodels* are also important.

B. Discussion of the Related Work

In general, static and symbolic analysis of software has difficulty capturing the actual, dynamic behavior; especially in the case of dynamic types (e.g., inheritance, dynamic binding, exception handling) and jumps. In these cases, it is often favorable to observe the actual system for the behavior. Since static techniques either unfold or do not explore function calls, they lack support for recursive behavior. In addition, because these techniques only look at static artifacts, they lack any form of timing or usage analysis.

In the area of dynamic analysis, the focus is on obtaining a rich but flat control flow model. A lot of effort has been put in enriching models with more accurate choice and loop information, guards, and other predicates. However, notions of recursion or preciseness of models, or application of these models, like for analysis, seems to be largely ignored. The few approaches that do touch upon performance or frequency analysis ([1], [17], [20]) do so with models lacking formal semantics or model quality guarantees.

In contrast to dynamic analysis techniques, grammar inference approaches are actively looking for repeating sub patterns (i.e., sources for hierarchies). The used grammars have a strong formal basis. However, in the grammar inference domain, abstract symbols are assumed as input, and the notion of branching behavior (e.g., loops) or analysis is lost.

In the area of process mining, numerous techniques have been proposed. These techniques have strong roots in Petri nets, model conversions, and alignment-based analysis [2], [35] Process mining techniques yield formal models directly usable for advanced performance, frequency and conformance analysis. There are only a few techniques in this domain that touch upon the subject of hierarchies. In [21], [25], a hierarchy of anonymous clusters is created based on behavioral abstractions. The hierarchy of anonymous clusters in [15] is based on functional and inclusion dependency discovery techniques over extra data in the event log. None of these techniques yields named submodels or supports recursion.

Process mining techniques rely on event logs for their input. These event logs can easily be obtained via the same techniques used by dynamic analysis for singular and distributed systems [34]. Example techniques include, but are not limited to, Java Agents, Javassist [13], [14], AspectJ [19], AspectC++ [42], AOP++ [57] and the TXL preprocessor [16].

III. DEFINITIONS

Before we explain the proposed discovery techniques, we first introduce the definitions for our input and internal representation. We start with some preliminaries in Subsection III-A. In Subsection III-B we introduce two types of event logs: "flat" event logs (our input), and hierarchical event logs (used during discovery). Finally, in Subsection III-C and III-D, we will discuss the process tree model and our novel extension: *hierarchical process trees*. Throughout this paper, we will be using the program given in Listing 1 as a running example, and assume we can log the start and end of each method.

```
1   public class Main {
2       public static void main(int argument) {
3           A inst = input(argument);
4           inst.process(argument);
5           output();
6       }
7       private static A input(int i) { ... }
8       private static void output() { ... }
9   }
10  class A {
11      public void process(int i) { ... }
12  }
13  class B extends A {
14      public void process(int i) {
15          if (i <= 0) {
16              super.process(i);
17          } else {
18              stepPre();
19              process(i - 1);
20              stepPost();
21          }
22      }
23      private void stepPre() { ... }
24      private void stepPost() { ... }
25  }
```

Listing 1. Running example program code, logged at the method level.

A. Preliminaries

1) Multisets: We denote the set of all multisets over some set A as $\mathcal{B}(A)$. Note that the ordering of elements in a multiset is irrelevant.

2) Sequences: Given a set X, a sequence over X of length n is denoted as $\langle a_1, \ldots, a_n \rangle \in X^*$. We define $\langle a_1, \ldots, a_n \rangle [i] = a_i$. The empty sequence is denoted as ε. Note that the ordering of elements in a sequence is relevant. We write $\cdot$ to denote sequence concatenation, for example: $\langle a \rangle \cdot \langle b \rangle = \langle a, b \rangle$, and $\langle a \rangle \cdot \varepsilon = \langle a \rangle$. We write $x \upharpoonright_i$ to denote sequence (tail) projection, where $\langle a_1, a_2, \ldots, a_n \rangle \upharpoonright_i = \langle a_i, \ldots, a_n \rangle$. For example: $\langle a, b \rangle \upharpoonright_0 = \langle a, b \rangle$, $\langle a, b \rangle \upharpoonright_1 = \langle b \rangle$, and $\langle a, b \rangle \upharpoonright_2 = \varepsilon$. We write $\diamond$ to denote sequence interleaving (shuffle). For example: $\langle a, b \rangle \diamond \langle c, d \rangle = \{ \langle a, b, c, d \rangle, \langle a, c, b, d \rangle, \langle a, c, d, b \rangle, \langle c, a, b, d \rangle, \langle c, a, d, b \rangle, \langle c, d, a, b \rangle \}$.

B. Event Logs

1) "Flat" Event Logs: The starting point for any process mining technique is an *event log*, a set of *events* grouped into *traces*, describing what happened when. Each trace corresponds to an execution of a process; typically representing an example run in a software context. Various *attributes* may characterize events, e.g., an event may have a timestamp, correspond to an activity, denote a start or end, reference a line number, is executed by a particular process or thread, etc.

2) Hierarchical Event Logs: A hierarchical event log extends on a "flat" event log by assigning multiple activities to events; each activity describes what happened at a different level of granularity. We assume a "flat" event log as input. Based on this input, we will create a hierarchical event log for

discovery. For the sake of clarity, we will ignore most event attributes, and use sequences of activities directly, as defined below.

Definition III.1 (Hierarchical Event Log):
Let $\mathbb{A}$ be a set of activities. Let $L \in \mathcal{B}((\mathbb{A}^*)^*)$ be a hierarchical event log, a multiset of traces. A trace $t \in L$, with $t \in (\mathbb{A}^*)^*$, is a sequence of events. Each event $x \in t$, with $x \in \mathbb{A}^*$, is described by a sequence of activities, stating which activity was executed at each level in the hierarchy. ⌟

Consider, for example, the hierarchical event log $L = [\langle\langle g, a\rangle, \langle g, b\rangle, \langle c\rangle\rangle]$. This log has one trace, where the first event is labeled $\langle g, a\rangle$, the second event is labeled $\langle g, b\rangle$, and the third event is labeled $\langle c\rangle$. For the sake of readability, we will use the following shorthand notation: $L = [\langle g.a, g.b, c\rangle]$. In this example log, we have two levels in our hierarchy: the longest event label has length 2, notation: $\|L\| = 2$. Complex behavior, like choices, loops and parallel (interleaved) behavior, is typically represented in an event log via multiple (sub)traces, showing the different execution paths.

We write the following to denote hierarchy concatenation: $f.\langle g.a, g.b, c\rangle = \langle f.g.a, f.g.b, f.c\rangle$. We generalize concatenation to hierarchical logs: $f.L = [f.t \mid t \in L]$.

We extend sequence projection to hierarchical traces and logs, such that a fixed length prefix is removed for all events: $\langle g.a, g.b, c\rangle \restriction_0^* = \langle g.a, g.b, c\rangle$, $\langle g.a, g.b, c\rangle \restriction_1^* = \langle a, b\rangle$, $\langle g.a, g.b, c\rangle \restriction_2^* = \varepsilon$. For logs: $L\restriction_i^* = [t\restriction_i^* \mid t \in L]$.

In Table II, an example hierarchical trace is shown. Here, we used the class plus method name as activities. While generating logs, one could also include the full package name (i.e., a canonical name), method parameter signature (to distinguish overloaded methods), and more.

C. Process Trees

In this subsection, we introduce *process trees* as a notation to compactly represent *block-structured models*. An important property of block-structured models is that they are *sound by construction*; they do not suffer from deadlocks, livelocks, and other anomalies. In addition, process trees are tailored towards process discovery and have been used previously to discover block-structured workflow nets [35]. A process tree describes a language; an operator describes how the languages of its subtrees are to be combined.

Definition III.2 (Process Tree):
We formally define *process trees* recursively. We assume a finite alphabet $\mathbb{A}$ of activities and a set $\bigotimes$ of operators to be given. Symbol $\tau \notin \mathbb{A}$ denotes the silent activity.
- a with $a \in (\mathbb{A} \cup \{\tau\})$ is a process tree;
- Let $P_1, \ldots, P_n$ with $n > 0$ be process trees and let $\otimes \in \bigotimes$ be a process tree operator, then $\otimes(P_1, \ldots, P_n)$ is a process tree.

We consider the following operators for process trees:
$\rightarrow$ denotes the *sequential execution* of all subtrees
$\times$ denotes the *exclusive choice* between one of the subtrees
$\circlearrowleft$ denotes the *structured loop* of loop body P_1 and alternative loop back paths $P_2, \ldots, P_n$ (with $n \geq 2$)
$\wedge$ denotes the *parallel (interleaved) execution* of all subtrees ⌟

To describe the semantics of process trees, the language of a process tree P is defined using a recursive monotonic function $\mathcal{L}(P)$, where each operator $\otimes$ has a language join function $\otimes^l$:

$$\mathcal{L}(a) = \{\langle a\rangle\} \textbf{ for } a \in \mathbb{A}$$
$$\mathcal{L}(\tau) = \{\varepsilon\}$$
$$\mathcal{L}(\otimes(P_1, \ldots, P_n)) = \otimes^l(\mathcal{L}(P_1), \ldots, \mathcal{L}(P_n))$$

Each operator has its own language join function $\otimes^l$. The language join functions below are borrowed from [35], [46]:

$$\rightarrow^l(L_1, \ldots, L_n) = \{t_1 \cdot \ldots \cdot t_n \mid \forall 1 \leq i \leq n : t_i \in L_i\}$$
$$\times^l(L_1, \ldots, L_n) = \bigcup_{1 \leq i \leq n} L_i$$
$$\circlearrowleft^l(L_1, \ldots, L_n) = \{t_1 \cdot t_1' \cdot t_2 \cdot t_2' \cdot \ldots \cdot t_{m-1} \cdot t_{m-1}' \cdot t_m$$
$$\mid \forall i : t_i \in L_1, t_i' \in \bigcup_{2 \leq j \leq n} L_j\}$$
$$\wedge^l(L_1, \ldots, L_n) = \{t' \in (t_1 \diamond \ldots \diamond t_n) \mid \forall 1 \leq i \leq n : t_i \in L_i\}$$

Example process trees and their languages:

$$\mathcal{L}(\rightarrow(a, \times(b, c))) = \{\langle a, b\rangle, \langle a, c\rangle\}$$
$$\mathcal{L}(\wedge(a, b)) = \{\langle a, b\rangle, \langle b, a\rangle\}$$
$$\mathcal{L}(\wedge(a, \rightarrow(b, c))) = \{\langle a, b, c\rangle, \langle b, a, c\rangle, \langle b, c, a\rangle\}$$
$$\mathcal{L}(\circlearrowleft(a, b)) = \{\langle a\rangle, \langle a, b, a\rangle, \langle a, b, a, b, a\rangle, \ldots\}$$

D. Hierarchical Process Trees

We extend the process tree representation to support hierarchical and recursive behavior. We add a new tree operator to represent a named submodel, and add a new tree leaf to denote a recursive reference. Figure 2 shows an example model.

Definition III.3 (Hierarchical Process Tree):
We formally define *hierarchical process trees* recursively. We assume a finite alphabet $\mathbb{A}$ of activities to be given.
- Any process tree is also a hierarchical process tree;
- Let P be a hierarchical process tree, and $f \in \mathbb{A}$, then $\nabla_f(P)$ is a hierarchical process tree that denotes the named subtree, with name f and subtree P;
- $\triangle_f$ with $f \in \mathbb{A}$ is a hierarchical process tree. Combined with a named subtree operator ∇_f, this leaf denotes the point where we recurse on the named subtree. See also the language definition below. ⌟

The semantics of hierarchical process trees are defined by extending the language function $\mathcal{L}(P)$. A recursion at a leaf $\triangle_f$ is 'marked' in the language, and resolved at the level of the first corresponding named submodel ∇_f. Function ψ_f^l 'scans' a single symbol and resolves for marked recursive calls (see $\psi_f^l(\triangle_f)$). Note that via ψ_f^l, the language ∇_f^l is defined recursively and yields a hierarchical, recursive language.

$$\mathcal{L}(\triangle_f) = \{\langle\triangle_f\rangle\} \textbf{ for } f \in \mathbb{A}$$
$$\nabla_f^l(L) = \{f.(t_1' \cdot \ldots \cdot t_n') \mid \langle x_1, \ldots, x_n\rangle \in L,$$
$$\forall 1 \leq i \leq n : t_i' \in \psi_f^l(x_i)\} \textbf{ for } f \in \mathbb{A}$$
$$\textbf{where } \psi_f^l(\varepsilon) = \{\varepsilon\}$$
$$\psi_f^l(\triangle_f) = \nabla_f^l(L) \textbf{ for } f \in \mathbb{A}$$
$$\psi_f^l(\triangle_g) = \{\langle\triangle_g\rangle\} \textbf{ for } g \neq f \wedge g \in \mathbb{A}$$
$$\psi_f^l(a.x) = \{a.(t') \mid t' \in \psi_f^l(x)\} \textbf{ for } a \in \mathbb{A}, x \in \mathbb{A}^*$$

Example hierarchical process trees and their languages:

$$\mathcal{L}(\nabla_f(\rightarrow(a,b)) = \{\, \langle\, f.a, f.b\,\rangle\,\}$$
$$\mathcal{L}(\nabla_f(\rightarrow(a, \nabla_g(b)))) = \{\, \langle\, f.a, f.g.b\,\rangle\,\}$$
$$\mathcal{L}(\nabla_f(\times(\rightarrow(a, \triangle_f), b))) = \{\, \langle\, f.b\,\rangle, \langle\, f.a, f.f.b\,\rangle,$$
$$\langle\, f.a, f.f.a, f.f.f.b\,\rangle, \dots\}$$
$$\mathcal{L}(\nabla_f(\nabla_g(\times(a, \triangle_f)))) = \{\, \langle\, f.g.a\,\rangle, \langle\, f.g.f.g.a\,\rangle,$$
$$\langle\, f.g.f.g.f.g.a\,\rangle, \dots\}$$
$$\mathcal{L}(\nabla_f(\nabla_g(\times(a, \triangle_f, \triangle_g)))) = \{\, \langle\, f.g.a\,\rangle, \langle\, f.g.g.a\,\rangle, \langle\, f.g.f.g.a\,\rangle,$$
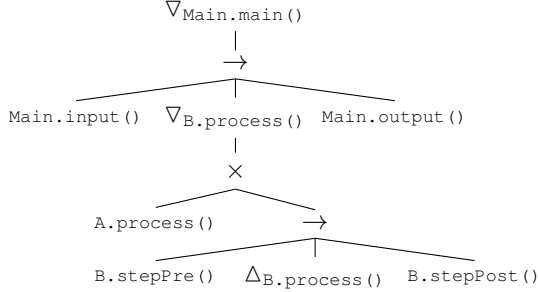$$\langle\, f.g.f.g.g.a\,\rangle, \langle\, f.g.g.f.g.a\,\rangle, \dots\}$$

Fig. 2. The hierarchical process tree model corresponding to the hierarchical event log with the trace from Table II. Note that we modeled the recursion at `B.process()` explicitly.

IV. HEURISTICS FOR HIERARCHY

To go from "flat" event logs (our input) to hierarchical event logs (used during discovery), we rely on transformation heuristics. In the typical software system case, we will use the *Nested Calls* heuristic, but our implementation also provides other heuristics for different analysis scenarios. We will discuss three of these heuristics for hierarchy next.

1) *Nested Calls* captures the "executions call stacks". Through the use of the life-cycle attribute (start-end), we can view flat event logs as a collection of intervals, and use *interval containment* to build our hierarchy. In Figure 3, a trace from the event log corresponding to the program in Listing 1 is depicted as intervals. Table II shows the corresponding "nested calls" hierarchical event log trace. 2) *Structured Names* captures the static structure or "architecture" of the software source code. By splitting activities like "`package.class.method()`" on "." into $\langle$ `package`, `class`, `method()` $\rangle$, we can utilize the designed hierarchy in the code as a basis for discovery. 3) *Attribute Combination* is a more generic approach, where we can combine several attributes associated with events in a certain order. For example, some events are annotated with a high-level and a low-level description, like an interface or protocol name plus a specific function or message/signal name.

V. MODEL DISCOVERY

A. Discovery Framework

Our techniques are based on the Inductive miner framework for discovering process tree models. This framework is described in [35]. Given a set $\bigotimes$ of process tree operators, [35]
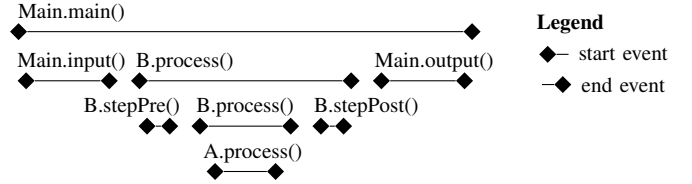
Fig. 3. A trace from the event log corresponding to logging the methods in the program in Listing 1, depicted as intervals. By using the *Nested Call* heuristics, a hierarchical event log can be constructed from contained intervals, see also Table II. For example, B.process() is contained in Main.main().

TABLE II
A SINGLE TRACE IN AN EXAMPLE EVENT LOG CORRESPONDING TO THE PROGRAM IN LISTING 1 AND THE INTERVALS IN FIGURE 3. EACH COLUMN IS ONE EVENT, AND EACH ROW A LEVEL IN THE HIERARCHY.

Main.main()	Main.main()	Main.main()	Main.main()	Main.main()
Main.input()	B.process()	B.process()	B.process()	Main.output()
		B.stepPre()	B.process()	B.stepPost()
			A.process()	

defines a framework to discover models using a divide and conquer approach. Given a log L, the framework searches for possible splits of L into sublogs $L_1, \dots, L_n$, such that these logs combined with an operator $\otimes$ can (at least) reproduce L again. It then recurses on the corresponding sublogs and returns the discovered submodels. Logs with empty traces or traces with a single activity form the base cases for this framework. Note that the produced model can be a generalization of the log; see for example the language of the structured loop ($\circlearrowleft$). In Table III, an example run of the framework is given.

TABLE III
EXAMPLE DISCOVERY ON THE LOG $[\,\langle a, b, d\,\rangle, \langle a, c, d, e, d\,\rangle\,]$. THE ROWS ILLUSTRATE HOW THE DISCOVERY PROGRESSES. THE HIGHLIGHTS INDICATE THE SUBLOGS USED, AND RELATE THEM TO THE CORRESPONDING PARTIAL MODEL THAT IS DISCOVERED.

We present two adaptations of the framework described above: one for hierarchy (*Naïve Discovery*, Subsection V-B), and one for recursions (*Recursion Aware Discovery*, Subsection V-C). Our adaptations maintain the termination guarantee, perfect fitness guarantee, language rediscoverability guarantee,

and the polynomial runtime complexity from [35]. The details of the above guarantees and runtime complexity for our adaptations are detailed in Subsection V-D. In our implementation, we rely on the (in)frequency based variant to enable the discovery of an 80/20 model, see also Subsection VI-A.

B. Naive Discovery

We generalize the above approach to also support hierarchy by providing the option to split hierarchical event logs and use hierarchical sequence projection.

Using our generalized discovery framework, we define a naive realization associated with the named subtree operator ∇_f, and a slightly modified base case. The details are given in Algorithm 1. In Table IV, an example run is given.

This algorithmic variant is best suited for cases where recursion does not make much sense, for example, when we are using a hierarchical event log based on structured package-class-method name hierarchies.

Algorithm 1: Naive Discovery (*Naive*)

Input: A hierarchical event log L
Output: A hierarchical process tree P such that L fits P
Description: Extended framework, using the named subtree operator ∇_f.
$Naive(L)$
(1) **if** $\forall t \in L : t = \varepsilon$
(2)   **return** τ // the log is empty or only contains empty traces
(3) **else if** $\exists f \in \mathbb{A} : \forall t \in L : t = \langle f \rangle$
(4)   **return** f // the log only has a single low-level activity
(5) **else if** $\exists f \in \mathbb{A} : (\forall x \in t \in L : x[1] = f) \wedge (\exists x \in t \in L : |x| > 1)$
(6)   // all events start with f, and there is a lower level in the hierarchy
(7)   **return** $\nabla_f(Naive(L|_1^{*}))$
(8) **else**
(9)   // normal framework cases, based on the tree semantics (Def. III.2)
(10)   Split L into into sublogs $L_1, \ldots, L_n$, such that:
(11) $\exists \otimes \in \{\rightarrow, \times, \circlearrowleft, \wedge\} : L \subseteq \otimes^l(L_1, \ldots, L_n)$ // see [35]
(12)   **return** $\otimes(Naive(L_1), \ldots, Naive(L_n))$

Below are some more example logs and the models discovered. Note that with this naive approach, the recursion in the last example is not discovered.

$$Naive([\langle f.a, f.b \rangle, \langle f.c \rangle]) = \nabla_f(Naive([\langle a, b \rangle, \langle c \rangle]))$$
$$= \nabla_f(\times(\rightarrow(a,b),c))$$
$$Naive([\langle f.a, f.g.f.b \rangle]) = \nabla_f(Naive([\langle a, g.f.b \rangle]))$$
$$= \nabla_f(\rightarrow(a, Naive([\langle g.f.b \rangle])))$$
$$= \nabla_f(\rightarrow(a, \nabla_g(\nabla_f(b))))$$
$$Naive([\langle f.a \rangle, \langle f \rangle]) = \nabla_f(Naive([\langle a \rangle, \varepsilon]))$$
$$= \nabla_f(\times(a, \tau))$$

C. Recursion Aware Discovery

In order to successfully detect recursion, we make some subtle but important changes. We rely on two key notions: 1) a *context path*, and 2) *delayed discovery*. Both are explained below, using the example shown in Table V.

This algorithmic variant is best suited for cases where recursion makes sense, for example, when we are using an event log based on the *Nested Calls* hierarchy (Section IV).

To detect recursion, we need to keep track of the named subtrees from the root to the current subtree. We call the sequence of activities on such a path the *context path*, notation $C \in \mathbb{A}^{*}$. The idea is that whenever we discover a named subtree ∇_f, and we encounter another activity f somewhere in the sublogs, we can verify this recursion using $f \in C$. Sublogs

collected during discovery are associated with a context path, notation $L(C)$. This approach is able to deal with complex recursions (see examples at the end) and overloaded methods (see the activity naming discussion in Section III-B2). In Table V, the current context path C at each step is shown.

TABLE IV
EXAMPLE NAIVE DISCOVERY ON THE LOG $[\langle f.a, f.f.b \rangle]$. THE ROWS ILLUSTRATE HOW THE DISCOVERY PROGRESSES. THE HIGHLIGHTS INDICATE THE SUBLOGS USED, AND RELATE THEM TO THE CORRESPONDING PARTIAL MODEL THAT IS DISCOVERED.

Step	Discovered Model	Event Log	Sublog View
1	∇_f | ?	$\begin{array}{cc} f & f \\ a & f \\ & b \end{array}$	$[\langle f.a, f.f.b \rangle]$
2	∇_f | $\rightarrow$ $a \quad \nabla_f$ | ?	$\begin{array}{cc} f & f \\ a & f \\ & b \end{array}$	$[\langle a, f.b \rangle]$
3	∇_f | $\rightarrow$ $a \quad \nabla_f$ | b	$\begin{array}{cc} f & f \\ a & f \\ & b \end{array}$	$[\langle b \rangle]$

TABLE V
EXAMPLE RECURSION AWARE DISCOVERY ON THE LOG $[\langle f.a, f.f.b \rangle]$. THE ROWS ILLUSTRATE HOW THE DISCOVERY PROGRESSES. THE HIGHLIGHTS INDICATE THE SUBLOGS USED, AND RELATE THEM TO THE CORRESPONDING PARTIAL MODEL THAT IS DISCOVERED.

Step	Discovered Model	Event Log	Sublog View
1	∇_f | ?	$\begin{array}{cc} f & f \\ a & f \\ & b \end{array}$	$L = [\langle f.a, f.f.b \rangle]$ (Context C $= \varepsilon$)
2	∇_f | $\rightarrow$ $a \quad \Delta_f$	$\begin{array}{cc} x & f \\ a & f \\ & b \end{array}$	$L(\langle f \rangle) = [\langle a, f.b \rangle]$ (Context C $= \langle f \rangle$)
3	∇_f | $\times$ $b \quad \rightarrow$ $a \quad \Delta_f$	$\begin{array}{cc} f & f \\ a & f \\ & b \end{array}$	$L(\langle f \rangle) = [\langle a, f.b \rangle, \langle b \rangle]$ (Context C $= \langle f \rangle$)

Let's have a closer look at steps 2 and 3 in Table V. Note how the same subtree is discovered twice. In step 2, we detect the recursion. And in step 3, we use the sublog after the recursion part as an additional set of traces. The idea illustrated here is that of *delayed discovery*. Instead of immediately discovering the subtree for a named subtree ∇_f, we delay that discovery. The corresponding sublog is associated with the current context path. For each context path, we discover a model for the associated sublog. During this discovery, the sublog associated with that context path may change. If that happens, we run that discovery again on the extended

sublog. Afterwards, we insert the partial models under the corresponding named subtrees operators.

Algorithm 2 details the recursion aware discovery algorithm; it uses Algorithm 3 for a single discovery run. In the example of Table V, we first discover on the complete log with the empty context path (Alg. 2, line 2). In step 1, we encounter the named subtree ∇_f, and associate $L(\langle f \rangle) = [\langle a, f.b \rangle]$, for context path $C = \langle f \rangle$ (Alg. 3, line 12). In step 2, we start discovery on $C = \langle f \rangle$ using the sublog $L(\langle f \rangle)$ (Alg. 2, line 5). In this discovery, we encounter the recursion $f \in C$, and add $[\langle b \rangle]$ to the sublog, resulting in $L(\langle f \rangle) = [\langle a, f.b \rangle, \langle b \rangle]$ (Alg. 3, line 8). Finally, in step 3, we rediscover for $C = \langle f \rangle$, now using the extended sublog (Alg. 2, line 5). In this discovery run, no sublog changes anymore. We insert the partial models under the corresponding named subtrees operators (Alg. 2, line 9) and return the result.

Algorithm 2: Recursion Aware Discovery (*RAD*)

Input: A hierarchical event log L
Output: A hierarchical process tree P such that L fits P
Description: Extended framework, using the named subtree and recursion operators.
$RAD(L)$
(1)   *// discover root model using the full event log ($C = \varepsilon$)*
(2) $root = RADrun(L, \varepsilon)$
(3)   *// discover the submodels using the recorded sublogs ($C \neq \varepsilon$)*
(4)   **Let** $model$ be an empty map, relating context paths to process trees
(5)   **while** $\exists C \in \mathbb{A}^* : L(C)$ changed **do** $model(C) = RADrun(L(C), C)$
(6)   *// glue the partial models $model(C)$ and root model $root$ together*
(7)   **foreach** node P in process tree $root$ (any order, including new children)
(8)   Let $C = \langle f \mid P' = \nabla_f \text{ **foreach** } P' \text{ on the path from } root \text{ to } P \rangle$
(9)   **if** $(\exists f : P = \nabla_f) \wedge C \in model$ **then** Set $model(C)$ as the child of P
(10) **return** $root$

Algorithm 3: Recursion Aware Discovery - single run

Input: A hierarchical event log L, and a context path C
Output: A hierarchical process tree P such that L fits P
Description: One single run/iteration in the RAD extended framework.
$RADrun(L, C)$
(1)   **if** $\forall t \in L : t = \varepsilon$
(2)   **return** τ *// the log is empty or only contains empty traces*
(3)   **else if** $\exists f \in \mathbb{A} : \forall t \in L : t = \langle f \rangle$
(4)   **return** f *// the log only has a single low-level activity*
(5)   **else if** $\exists f \in C : \forall x \in t \in L : x[1] = f$
(6)   *// recursion on f is detected*
(7) $C' = C_1 \cdot \langle f \rangle$ **where** $(C_1 \cdot \langle f \rangle \cdot C_2) = C$
(8) $L(C') = L(C') \cup L\!\upharpoonright_1^*$ *// $L\!\upharpoonright_1^*$ is added to the sublog for C'*
(9)   **return** $\triangle_f$
(10) **else if** $\exists f \in \mathbb{A} : (\forall x \in t \in L : x[1] = f) \wedge (\exists x \in t \in L : |x| > 1)$
(11) *// discovered a named subtree f, note that $f \notin C$ since line 5 was false*
(12) $L(C \cdot \langle f \rangle) = L\!\upharpoonright_1^*$ *// $L\!\upharpoonright_1^*$ is associated with $C' = C \cdot \langle x \rangle$*
(13) **return** ∇_f
(14) **else**
(15) *// normal framework cases, based on the tree semantics (Def. III.2)*
(16) Split L into into sublogs $L_1, \ldots, L_n$, such that:
(17) $\exists \otimes \in \{\rightarrow, \times, \circlearrowleft, \wedge\} : L \subseteq \otimes^l(L_1, \ldots, L_n)$ *// see [35]*
(18) **return** $\otimes(RADrun(L_1, C), \ldots, RADrun(L_n, C))$

Below are some more example logs, the models discovered, and the sublogs associated with the involved context paths. Note that with this approach, complex recursions are also discovered.

$$RAD([\langle f.a, f.g.f.b \rangle]) = \nabla_f(\times(b, \rightarrow(a, \nabla_g(\triangle_f))))$$
$$\textbf{where } L(\langle f \rangle) = [\langle b \rangle, \langle a, g.f.b \rangle]$$
$$L(\langle f, g \rangle) = [\langle f.b \rangle]$$
$$RAD([\langle f.g.g.a \rangle, \langle f.g.f.g.a \rangle]) = \nabla_f(\nabla_g(\times(a, \triangle_f, \triangle_g)))$$
$$\textbf{where } L(\langle f \rangle) = [\langle g.g.a \rangle, \langle g.f.g.a \rangle, \langle g.a \rangle]$$
$$L(\langle f, g \rangle) = [\langle g.a \rangle, \langle f.g.a \rangle, \langle a \rangle]$$
$$RAD([\langle f.f \rangle]) = \nabla_f(\times(\triangle_f, \tau))$$
$$\textbf{where } L(\langle f \rangle) = [\langle f \rangle, \varepsilon]$$

D. Termination, Perfect Fitness, Language Rediscoverability, and Runtime Complexity

Our Naïve Discovery and Recursion Aware Discovery adaptations of the framework described [35] maintains the termination guarantee, perfect fitness guarantee, language rediscoverability guarantee, and the polynomial runtime complexity. We will discuss each of these properties using the simplified theorems and proofs from [36].

1) Termination Guarantee: The termination guarantee is based on the proof for [36, Theorem 2, Page 7]. The basis for the termination proof relies on the fact that the algorithm only performs finitely many recursions. For the standard process tree operators in the original framework, it is shown that the log split operator only yields finitely many sublogs. Hence, for our adaptations, we only have to show that the new hierarchy and recursion cases only yield finitely many recursions.

Theorem V.1: Naïve Discovery terminates.

Proof: Consider the named subtree case on Algorithm 1, line 7. Observe that the log L has a finite depth, i.e., a finite number of levels in the hierarchy. Note that the sequence projection $L\!\upharpoonright_1^*$ yields strictly smaller event logs, i.e, the number of levels in the hierarchy strictly decreases. We can conclude that the named subtree case for the Naïve Discovery yields only finitely many recursions. Hence, the Naïve Discovery adaptation maintains the termination guarantee of [36, Theorem 2, Page 7]. ∎

Theorem V.2: Recursion Aware Discovery terminates.

Proof: Consider the named subtree and recursion cases in Algorithm 3 on lines 8 and 12. Note that, by construction, for all the cases where we end up in Algorithm 3, line 8, we know that L is derived from, and bounded by, $L(C')$ as follows: $L \subseteq \{ L'\!\upharpoonright_i^* \mid L' \subseteq L(C') \wedge 0 \leq i \leq \|L(C')\| \}$. Observe that the log L has a finite depth, i.e., a finite number of levels in the hierarchy. Note that the sequence projection $L\!\upharpoonright_1^*$ yields strictly smaller event logs, i.e, the number of levels in the hierarchy strictly decreases. Hence, we can conclude that $L(C')$ only changes finitely often. Since C is derived from the log depth, we also have a finitely many sublogs $L(C')$ that are being used. Hence, the loop on Algorithm 2, line 5 terminates, and thus the Recursion Aware Discovery adaptation maintains the termination guarantee of [36, Theorem 2, Page 7]. ∎

2) Perfect Fitness: As stated in the introduction, we want the discovered model to fit the actual behavior. That is, we want the discovered model to at least contain all the behavior in the event log. The perfect fitness guarantee states that all the log behavior is in the discovered model, and we proof this using the proof for [36, Theorem 3, Page 7]. The fitness proof is based on induction on the log size[1]. As induction hypothesis, we assume that for all sublogs, the discovery framework returns a fitting model, and then prove that the step maintains this property. That is, for all sublogs L' we have a corresponding submodel P' such that $L' \subseteq \mathcal{L}(P')$. For

---

[1]Formally, the original induction is on the log size plus a counter parameter. However, for our proofs, we can ignore this counting parameter.

our adaptations, it suffices to show that the named subtree and recursion operators do not violate this assumption.

Theorem V.3: Naïve Discovery returns a process model that fits the log.

Proof: By simple code inspection on Algorithm 1, line 7 and using the induction hypothesis on $L|_1^*$, we can see that for the named subtree operator we return a process model that fits the log L. Since this line is the only adaptation, the Naïve Discovery adaptation maintains the perfect fitness guarantee of [36, Theorem 3, Page 7]. ∎

Theorem V.4: Recursion Aware Discovery returns a process model that fits the log.

Proof: Consider the named subtree case on Algorithm 3, line 12. Using the induction hypothesis on $L(\mathrm{C} \cdot \langle f \rangle) = L|_1^*$, we know that $model(\mathrm{C} \cdot \langle f \rangle)$ will fit $L(\mathrm{C} \cdot \langle f \rangle)$. By Algorithm 2, line 9, we know that $model(\mathrm{C} \cdot \langle f \rangle)$ will be the child of ∇_f. Hence, for the named subtree operator we return a process model that fits the log L.

Consider the recursion case on Algorithm 3, line 8. Since $f \in \mathrm{C}$, we know there must exist a named subtree ∇_f corresponding to the recursive operator $\triangle_f$. Due to Algorithm 2, line 5 and the induction hypthesis, we know that at the end $model(\mathrm{C}')$ fits $L(\mathrm{C}')$ (i.e., $L(\mathrm{C}') \subseteq \mathcal{L}(model(\mathrm{C}'))$). Since, by construction, we know $L|_1^* \subseteq L(\mathrm{C}')$, $model(\mathrm{C}')$ also fits $L|_1^*$. By Algorithm 2, line 9, we know that $\triangle_f$ will be in the subtree of ∇_f. Hence, for the recursion operator we return a process model that fits the log L.

We conclude that the Recursion Aware Discovery adaptation maintains the perfect fitness guarantee of [36, Theorem 3, Page 7]. ∎

3) Language Rediscoverability: The language rediscoverability property tells whether and under which conditions a discovery algorithm can discover a model that is language-equivalent to the original process. That is, given a 'system model' P and an event log L that is complete w.r.t. P (for some notion of completeness), then we rediscover a model P' such that $\mathcal{L}(P') = \mathcal{L}(P)$.

We will show language rediscoverability in several steps. First, we will define the notion of language complete logs. Then, we define the class of models that can be language-rediscovered. And finally, we will detail the language rediscoverability proofs.

a) Language Completeness: Language rediscoverability holds for directly-follows complete logs. We adapt this notion of directly-folllows completeness from [36] by simply applying the existing definition to hierarchical event logs:

Definition 1 (Directly-follows completeness): Let $Start(L)$ and $End(L)$ denote the set of start and end symbols amongst all traces, respectively. A log L is directly-follows complete to a model P, denoted as $L \diamond_{df} P$, iff:
1) $\langle \ldots, x, y, \ldots \rangle \in \mathcal{L}(P) \Rightarrow \langle \ldots, x, y, \ldots \rangle \in L$;
2) $Start(\mathcal{L}(P)) \subseteq Start(L)$;
3) $End(\mathcal{L}(P)) \subseteq End(L)$; and
4) $\Sigma(P) \subseteq \Sigma(L)$.

Note that directly-follows completeness is defined over all levels of a hierarchical log.

b) Class of Language-Rediscoverable Models: We will prove language rediscoverability for the following class of models. Let $\Sigma(P)$ denote the set of activities in P. A model P is in the class of language rediscoverable models iff for all nodes $\otimes(P_1, \ldots, P_n)$ in P we have:
1) No duplicate activities: $\forall i \neq j : \Sigma(P_i) \cap \Sigma(P_j) = \emptyset$;
2) In the case of a loop, the sets of start and end activities of the first branch must be disjoint:
   $\otimes = \circlearrowright \Rightarrow Start(\mathcal{L}(P_1)) \cap End(\mathcal{L}(P_1)) = \emptyset$
3) No taus are allowed: $\forall i \leq n : P_i \neq \tau$;
4) In the case of a recursion node $\triangle_f$, there exists a corresponding named subtree node ∇_f on the path from P to $\triangle_f$.

Note that the first three criteria follow directly from the language rediscoverability class from [36]. The last criteria is added to have well-defined recursions in our hierarchical process trees.

c) Language-Rediscoverable Guarantee: The language rediscoverability guarantee is based on the proof for [36, Theorem 14, Page 16]. The proof in [36] is based on three lemmas:
- [36, Lemma 11, Page 15] guarantees that any root process tree operator is rediscovered;
- [36, Lemma 12, Page 16] guarantees that the base cases can be rediscovered; and
- [36, Lemma 13, Page 16] guarantees that for all process tree operators the log is correctly subdivided.

For our adaptations, we have to show:
1) Our recursion base case maintains [36, Lemma 12, Page 16]; and
2) Our named subtree operator maintains [36, Lemma 11, Page 15] and [36, Lemma 13, Page 16].

Theorem V.5: Naïve Discovery preserves language rediscoverability.

Proof: We only have to show that the introduction of the named subtree operator maintains language rediscoverability.

First, we show for the named subtree operator that the root process tree operator is rediscovered (Lemma 11). Assume a process tree $P = \nabla_f(P_1)$, for any $f \in \mathbb{A}$, and let L be a log such that $L \diamond_{df} P$. Since we know that $L \diamond_{df} P$, we know that $\forall x \in t \in L : x[1] = f$, and there must be a lower level in the tree. By simple code inspection on Algorithm 1, line 7, we can see that the Naïve Discovery will yield ∇_f.

Next, we show for the named subtree operator that the log is correctly subdivided (Lemma 13). That is, lets assume: 1) a model $P = \nabla_f(P_1)$ adhering to the model restrictions; and 2) $L \subseteq \mathcal{L}(P) \wedge L \diamond_{df} P$. Then we have to show that any sublog L_i we recurse upon has: $L_i \subseteq \mathcal{L}(P_i) \wedge L_i \diamond_{df} P_i$. For the named subtree operator, we have exactly one sublog we recurse upon: $L_1 = L|_1^*$. We can easily prove this using the sequence projection on the induction hypothesis: $L|_1^* \subseteq \mathcal{L}(P)|_1^*$, after substitution: $L|_1^* \subseteq \mathcal{L}(\nabla_f(P_1))|_1^*$. By definition of the semantics for ∇_f, we can rewrite this to: $L|_1^* \subseteq \mathcal{L}(P_1)$. The proof construction for $L_i \diamond_{df} P_i$ is analogous. Hence, for the named subtree operator that the log is correctly subdivided.

We can conclude that the Naïve Discovery adaptation preserves language rediscoverability guarantee of [36, Theorem 14, Page 16]. ∎

Theorem V.6: Recursion Aware Discovery preserves language rediscoverability.

Proof: The proof for the introduction of the named subtree operator is analogous to the proof for Theorem V.5, using the fact that always $L\!\restriction_1^* \subseteq L(\mathrm{C}')$ for the corresponding context path C'.

We only have to show that the introduction of the recursion operator maintains language rediscoverability (Lemma 12). That is, assume: 1) a model $P = \triangle_f$ adhering to the model restrictions; and 2) $L \subseteq \mathcal{L}(P) \wedge L \diamond_{df} P$. Then we have to show that we discover the model P' such that $P' = P$.

Since we adhere to the model restrictions, due to restriction 4, we know there must be a larger model P'' such that the recursion node $\triangle_f$ is a leaf of P'' and there exists a corresponding named subtree node $\triangledown_f$ on the path from P'' to $\triangle_f$. Thus, we can conclude that L must be the sublog associated with a context path C such that $f \in \mathrm{C}$. By code inspection on Algorithm 3, line 5, we see that we only have to prove that $\forall x \in t \in L : x[1] = f$. This follows directly from $L \diamond_{df} P$. Hence, the recursion operator is correctly rediscovered.

We can conclude that the Recursion Aware Discovery adaptation preserves language rediscoverability guarantee of [36, Theorem 14, Page 16]. ∎

4) Runtime Complexity: In [36, Run Time Complexity, Page 17], the authors describe how the basic discovery framework is implemented as a polynomial algorithm. For the selection and log splitting for the normal process tree operators (Alg. 1, line 10, and Alg. 3, line 16), existing polynomial algorithms were used. Furthermore, for the original framework, the number of recursions made is bounded by the number of activities: $O(|\Sigma(L)|)$. We will show that this polynomial runtime complexity is maintained for our adaptations.

In our Naïve Discovery adaptation, the number of recursions is determined by Algorithm 1, lines 7 and 10. For line 7, the number of recursions is bounded by the depth of the hierarchical event log: $O(\|L\|)$. For line 10, the original number of activities bound holds: $O(|\Sigma(L)|)$. Thus, the total number of recursions for our Naïve Discovery is bounded by $O(\|L\| + |\Sigma(L)|)$. Hence, the Naïve Discovery adaptation has a polynomial runtime complexity.

In one run of our Recursion Aware Discovery, the number of recursions is determined by Algorithm 3, line 16. Note that the recursion and named subtree cases do not recurse directly due to the delayed discovery principle. For line 16, the original number of activities bound holds: $O(|\Sigma(L)|)$. Thus, we can conclude that Algorithm 3 has a polynomial runtime complexity.

For the complete Recursion Aware Discovery, the runtime complexity is determined by Algorithm 2, lines 5 and 9. Each iteration of the loop at line 5 is polynomial. The number of iterations is determined by the number of times an $L(\mathrm{C})$ is changed. Based on Algorithm 3, lines 8 and 12, the number

of times an $L(\mathrm{C})$ is changed is bounded by the depth of the hierarchical event log: $O(\|L\|)$. Thus, the total number of iterations is polynomial and bounded by $O(\|L\|)$. Each iteration of the loop at line 9 is polinomial in the named tree depth, and thus bounded by $O(\|L\|)$. The number of iterations is determined by the number of named subtrees, and thus also bounded by $O(\|L\|)$. Hence, the Recursion Aware Discovery adaptation has a polynomial runtime complexity.

VI. Using and Visualizing the Discovered Model

Discovering a behavioral backbone model is only step one. Equally important is how one is going to use the model, both for analysis and for further model driven engineering. In this section, we touch upon some of the solutions we implemented, and demo in Subsection VII-C and Figure 6.

A. Rewriting, Filtering and the 80/20 Model

To help the user understand the logged behavior, we provide several ways of filtering the model, reducing the visible complexity, and adjusting the model quality.

Based on frequency information, we allow the user to inspect an 80/20 model. An 80/20 model describes the mainstream (80%) behavior using a simple (20%) model [35]. We allow the user to interactively select the cutoff (80% by default) using sliders directly next to the model visualization, thus enabling the "real-time exploration" of behavior. Unusual behavior can be projected and highlighted onto the 80% model using existing conformance and deviation detection techniques [2]. This way, it is immediately clear where the unusual behavior is present in the model, and how it is different from the mainstream behavior.

Based on hierarchical information, we allow both coarse and fine grained filtering. Using sliders, the user can quickly select a minimum and maximum hierarchical depth to inspect, and hide other parts of the model. The idea of depth filtering is illustrated in Figure 4. Afterwards, users can interactively fold and unfold parts of the hierarchy. By searching, users can quickly locate areas of interest. Using term-based tree rewriting (see Table VI), we present the user with a simplified model that preserves behavior.

TABLE VI
REDUCTION RULES FOR (HIERARCHICAL) PROCESS TREES

$\otimes(P_1) = P_1$	**for** $\otimes \in \{\rightarrow, \times, \wedge\}$
$\otimes(\ldots_1, \otimes(\ldots_2), \ldots_3) = \otimes(\ldots_1, \ldots_2, \ldots_3)$	**for** $\otimes \in \{\rightarrow, \wedge\}$
$\otimes(\ldots_1, \tau, \ldots_2) = \otimes(\ldots_1, \ldots_2)$	**for** $\otimes \in \{\rightarrow, \wedge\}$
$\times(\ldots_1, \tau, \ldots_2) = \times(\ldots_1, \ldots_2)$	**if** $\varepsilon \in \mathcal{L}(\ldots_1 \cup \ldots_2)$

B. Linking the Model to Event Data and the Source Code

For further analysis, like performance (timing), frequency (usage) and conformance, we annotate the discovered model with additional information. By annotating this information onto (parts of) the model, we can visualize it in the context of the behavior. This annotation is based on the event log data provided as input and is partly provided by existing algorithms in the Process Mining Toolkit ProM. Most notably, we 1) align the discovered model with the event log, as described in [2],
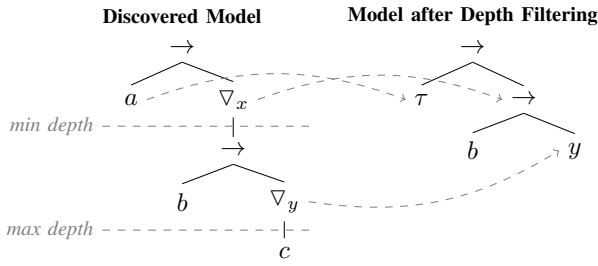
Discovered Model   **Model after Depth Filtering**

Fig. 4. Illustration of depth filtering, where we hide everything above x and below y. The dashed arrows relate the altered nodes.

(a) Hierarchical process tree

(b) Statechart

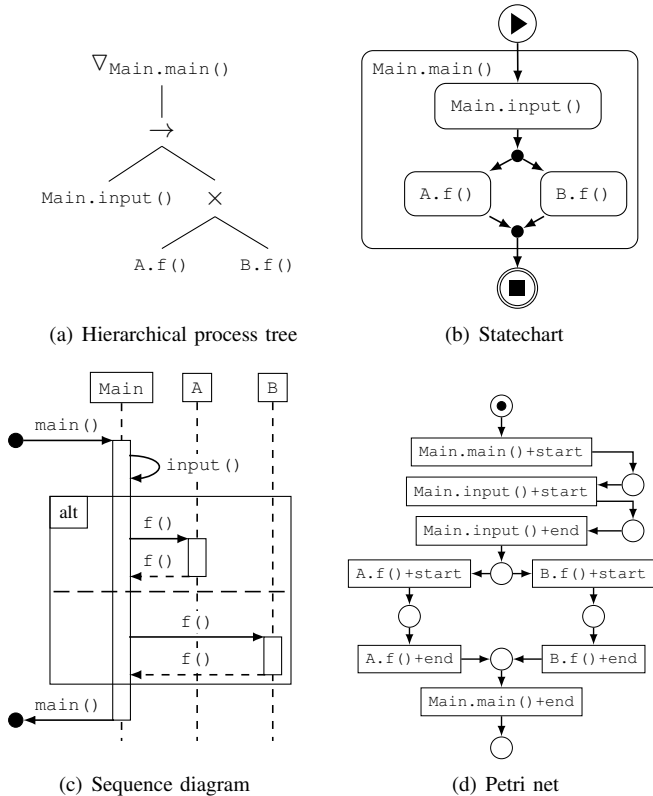(c) Sequence diagram

(d) Petri net

Fig. 5. A hierarchical process tree, and its mapping to different formalisms.

and 2) link model elements back to the referenced source code lines in Eclipse that generated the logged events.

C. Mapping and Visualizing

For visualization and integration with existing techniques, we implemented mappings to other formalisms. As noted in [35], process trees represent a "block-structured" language. Hence, we can simply map each operator to a block-structured concept in the target formalism, and preserve behavior by construction. We support mappings to formalisms such as Statecharts, (Data) Petri nets, Sequence diagrams, and BPMN diagrams. Some example mappings are given in Figure 5.

VII. EVALUATION

In this section, we compare our technique against related, implemented techniques. The proposed algorithms are implemented in the *Statechart* plugin for the process mining framework ProM [30]. In the remainder of this section, we will refer to Algorithm 1 as *Naïve*, and to Algorithm 2 as *RAD* (short for Recursion Aware Discovery). We end the evaluation by showing example results in our tool.

A. Input and Methodology for Comparative Evaluation

In the comparative evaluation, we focus on the quantitative aspects of the design criteria from the introduction. That is, the approach and tools should provide a behavioral backbone model that is precise and fits the actual system. We measure two aspects for a number of techniques and input event logs: 1) the *running time* of the technique, and 2) the *model quality*.

For the running time, we measured the average running time and associated 95% confidence interval over 30 micro-benchmark executions, after 10 warmup rounds for the Java JVM. Each technique is allowed at most 30 seconds for completing a single model discovery. For the model quality, we use fitness and precision as described in [2], and set a time limit of at most 5 minutes. In short, *fitness* expresses the part of the log that is represented by the model; *precision* expresses the behavior in the model that is present in the log. For these experiments we used a laptop with an i7-4700MQ CPU @ 2.40 GHz, Windows 8.1 and Java SE 1.7.0 67 (64 bit) with 12 GB of allocated RAM.

We selected five event logs as experiment input, covering a range of input problem sizes. The input problem size is typically measured in terms of four metrics: number of traces, number of events, number of activities (size of the alphabet), and average trace length. The event logs and their sizes are shown in Table VII. The *BPIC 2012* [51] and *BPIC 2013* [43] event logs are so called BPI Challenge logs. These large real-life event logs with complex behavior are often used in process mining evaluations. The challenge logs are made available yearly in conjunction with the BPM conference and are considered sufficiently large and complex inputs to stress test process mining techniques. The *JUnit 4.12* [31], Apache Commons *Crypto 1.0.0* [32], and *NASA CEV* [33] event logs are created using an extended version of the instrumentation tool developed for [34], yielding XES event logs with method-call level events. The JUnit 4.12 software [18] was executed once, using the example input found at [9]. For the Apache Commons Crypto 1.0.0 software [5], we executed the `CbcNoPaddingCipherStreamTest` unit test. For the NASA CEV software [37], we executed a unit test generated from the source code, covering all of the code branches.

TABLE VII
THE EVENT LOGS USED IN THE EVALUATION, WITH INPUT SIZES

Event Log	# Traces	# Events	# Acts	Avg. |Trace|
[51] BPIC 2012	13,087	262,200	24	20.04
[43] BPIC 2013	7,554	65,533	13	8.70
[31] JUnit 4.12	1	946	182	946.00
[32] Crypto 1.0.0	3	241,973	74	80,657.67
[33] NASA CEV	2,566	73,638	47	28.70

We compare our discovery algorithms against a selection of the techniques mentioned in Section II. Unfortunately, we could not compare against some of the related work due to invalid input assumptions or the lack of a reference implementa-

Algorithm (Heuristic)	Paths	BPIC 2012	BPIC 2013	JUnit 4.12	Crypto 1.0.0	NASA CEV
[49] Alpha miner		150.1	73.5	9.2	183.1	37.8
[56] Heuristics		840.2	278.0	1349.7	—^{T}	359.6
[21] Fuzzy miner		2858.5	827.4	166.8	—^{T}	4148.2
[50] ILP miner		—^{T}	6023.8	—^{T}	—^{T}	—^{T}
[52] ILP with filtering		7234.3	4354.7	—^{T}	—^{T}	—^{T}
[55] MINT, redblue, k=1		—^{T}	—^{T}	243.9	—^{T}	13426.0
[55] MINT, redblue, k=2		—^{T}	—^{T}	582.0	—^{T}	22213.4
[55] MINT, redblue, k=3		—^{T}	—^{T}	751.8	—^{T}	—^{T}
[55] MINT, ktails, k=1		—^{T}	—^{T}	108.9	—^{T}	—^{T}
[55] MINT, ktails, k=2		—^{T}	—^{T}	371.6	—^{T}	—^{T}
[55] MINT, ktails, k=3		—^{T}	—^{T}	512.3	—^{T}	—^{T}
[7] Synoptic		—^{T}	—^{T}	—^{T}	—^{T}	—^{T}
[35] IM (baseline)	1.0	3239.9	1351.2	215.7	10866.1	911.3
[35] IM (baseline)	0.8	5111.6	947.4	268.5	5213.9	912.6
Our techniques — Naïve (no heuristic)	1.0	3588.1	1117.8	278.3	26804.7	959.5
Naïve (no heuristic)	0.8	2865.6	603.4	298.5	—^{T}	1047.1
Naïve (Nested Calls)	1.0	n/a	n/a	15.1	1544.6	355.9
Naïve (Nested Calls)	0.8	n/a	n/a	12.6	1545.6	341.5
RAD (Nested Calls)	1.0	n/a	n/a	16.4	2186.4	439.0
RAD (Nested Calls)	0.8	n/a	n/a	14.6	2082.8	373.8
Naïve (Struct. Names)	0.8	2058.6	891.1	23.3	—^{M}	1275.9
RAD (Struct. Names)	0.8	2395.0	1028.6	23.7	—^{M}	1452.3

Avg. runtime (in milliseconds, with log scale plot), over 30 runs, with 95% confidence interval

^{M} Out of memory exception (12 GB) ^{T} Time limit exceeded (30 sec.) $^{n/a}$ Heuristic not applicable

tion. The *Inductive miner* (*IM*) [35] is our baseline comparison algorithm, since our approach builds upon the IM framework. For the Inductive miner and our derived techniques, we also consider the *paths* setting. This is the frequency cutoff for discovering an 80/20 model (see Subsection VI-A): 1.0 means all behavior, 0.8 means 80% of the behavior. The work of [49], [56], [52], [50] provides a comparison with standard process mining techniques. The *Fuzzy miner* [21] provides a comparison for hierarchy discovery. However, it yields models without semantics, and hence the quality cannot be determined. The *MINT algorithm (EFSM inference)* [55], [54] provides a comparison with the well-known *redblue* and *ktails* dynamic analysis techniques. The *Synoptic* algorithm [7], [6] is an invariant based FSM inferrer.

For our techniques, we also consider the following *heuristics for hierarchy*: 1) No heuristic (use log as is), 2) Nested Calls, and 3) Structured `package.class.method` Names (Struct. Names). Note that the Nested Calls heuristic is only applicable for software event logs, and not for the BPIC logs.

B. Comparative Evaluation Results and Discussion

1) Runtime Analysis: In Table VIII, the results for the runtime benchmark are given. As noted before, the Nested Calls setup is not applicable for the BPIC logs. We immediately observe that the ILP, MINT, and Synoptic algorithms could not finish in time on most logs. MINT and Synoptic have difficulty handling a large number of traces. We also notice that most setups require a long processing time and a lot of memory for the Apache Crypto log. Large trace lengths, such as in the Crypto log, are problematic for all approaches. Our techniques overcome this problem by using the hierarchy to divide large traces into multiple smaller traces (see below).

When we take a closer look at the actual running times, we observe the advantages of the heuristics for hierarchy and accompanied discovery algorithms. In all cases, using the right heuristic before discovery improves the running time. In extreme cases, like the Apache Crypto log, it even makes the difference between getting a result and having no result at all. Note that, with a poorly chosen heuristic, we might not get any improvements, e.g., note the absence of models for the Apache Crypto plus Structured Names heuristics.

The speedup factor for our technique depends on an implicit input problem size metric: the depth of the discovered hierarchy. In Table IX, the discovered depths are given for comparison. For example, the dramatic decrease in running time for the JUnit log can be explained by the large depth in hierarchy: 25 levels in this case. This implies that the event log is decomposed into many smaller sublogs, as per Alg. 1, line 7, and Alg. 3, lines 8 and 12. Hence, the imposed hierarchy indirectly yields a good decomposition of the problem, aiding the divide and conquer tactics of the underlying algorithms.

Event Log	Nested Calls		Struct. Names	
	Naïve	RAD	Naïve	RAD
[51] BPIC 2012	n/a	n/a	2	2
[43] BPIC 2013	n/a	n/a	2	2
[31] JUnit 4.12	25	18	9	9
[32] Apache Crypto 1.0.0	8	8	n/a	n/a
[33] NASA CEV	3	3	3	3

$^{n/a}$ No model (see Table VIII)

TABLE X
MODEL QUALITY SCORES FOR THE DIFFERENT ALGORITHMS, HIERARCHY HEURISTICS, PATHS FREQUENCY FILTER SETTINGS, AND EVENT LOGS.
SCORES RANGE FROM 0.0 TO 1.0, HIGHER IS BETTER. NO SCORES ARE AVAILABLE FOR THE FUZZY MODEL AND STRUCTURED NAMES HIERARCHY.

Algorithm (Heuristic)	Paths	BPIC 2012		BPIC 2013		JUnit 4.12		Crypto 1.0.0		NASA CEV	
		Fitness	Precision	Fitness	Precision	Fitness	Precision	Fitness	Precision	Fitness	Precision
[49] Alpha miner		_U	_U	0.36	0.88	_U	_U	_U	_U	0.91	0.06
[56] Heuristics		0.72	0.95	_U	_U	_U	_U	n/a	n/a	_U	_U
[50] ILP miner		n/a	n/a	1.00	0.36	n/a	n/a	n/a	n/a	n/a	n/a
[52] ILP, filtering		0.74	0.28	0.95	0.45	n/a	n/a	n/a	n/a	n/a	n/a
[55] MINT, redblue, k=1		n/a	n/a	n/a	n/a	0.00	_R	n/a	n/a	0.79	0.44
[55] MINT, redblue, k=2		n/a	n/a	n/a	n/a	0.48	0.17	n/a	n/a	0.81	0.45
[55] MINT, redblue, k=3		n/a	n/a	n/a	n/a	0.13	0.06	n/a	n/a	n/a	n/a
[55] MINT, ktails, k=1		n/a	n/a	n/a	n/a	0.00	_R	n/a	n/a	n/a	n/a
[55] MINT, ktails, k=2		n/a	n/a	n/a	n/a	0.43	0.16	n/a	n/a	n/a	n/a
[55] MINT, ktails, k=3		n/a	n/a	n/a	n/a	0.12	0.06	n/a	n/a	n/a	n/a
[7] Synoptic		n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
[35] IM (baseline)	1.0	1.00	0.37	1.00	0.62	1.00	0.34	1.00	0.35	1.00	0.55
[35] IM (baseline)	0.8	0.98	0.49	0.95	0.64	0.90	0.30	0.88	0.41	0.91	0.53
Our techniques — Naïve (no heuristic)	1.0	1.00	0.37	1.00	0.62	1.00	0.34	1.00	0.35	1.00	0.55
Naïve (no heuristic)	0.8	0.98	0.49	0.95	0.64	0.90	0.30	0.88	0.41	0.91	0.53
Naïve (Nested Calls)	1.0	n/a	n/a	n/a	n/a	1.00	0.84	1.00	0.45	1.00	0.80
Naïve (Nested Calls)	0.8	n/a	n/a	n/a	n/a	0.90	0.87	0.99	0.45	1.00	0.81
RAD (Nested Calls)	1.0	n/a	n/a	n/a	n/a	1.00	0.83	1.00	0.45	1.00	0.80
RAD (Nested Calls)	0.8	n/a	n/a	n/a	n/a	0.89	0.84	0.99	0.45	1.00	0.81

[T] Time limit exceeded (5 min.)   [R] Not reliable (fitness = 0)   [U] Unsound model   [n/a] No model (see Table VIII)

2) Model Quality Analysis: In Table X, the results of the model quality measurements are given. The Fuzzy miner is absent due to the lack of semantics for fuzzy models. For the Structured Names heuristic, there was no trivial event mapping for the quality score measurement.

Compare the quality scores between the *Nested Calls* setup with *No heuristics* and the related work. Note that in all cases, the Nested Calls setup yields a big improvement in precision, with no significant impact on fitness. The drop in quality for the JUnit - RAD case is due to the limitations in translation of the recursive structure to the input format of [2]. In addition, our technique without heuristics, with paths at 1.0, maintains the model quality guarantees (perfect fitness). Overall, we can conclude that the added expressiveness of modeling the hierarchy have a positive impact on the model quality.

Fig. 6. Screenshot of the workbench tool implementation [30]

C. Using the Tool

In Figure 6, we show some of the functionality and solutions available via the workbench tool implementation in ProM [30]. Via the user interface, the analyst first selects the desired heuristic for hierarchy. Afterwards, the analysis is presented through a Statechart visualization of the discovered model, annotated with frequency information. With the sliders on the right and the tree view and search box on the left, the analyst can adjust parameters in real time and interactively explore the model. The analyst can switch visualizations directly in the workbench UI, and view the model as, for example, a sequence diagram. Thanks to a corresponding Eclipse plugin, the model and analysis results can be linked back to the source code. A simple double-click on the model allows for a jump to the corresponding source code location. In addition, we can overlay the code in Eclipse with our model analysis results.

VIII. CONCLUSION AND FUTURE WORK

In this paper, *we 1) proposed a novel hierarchy and recursion extension to the process tree model; and 2) defined the first, recursion aware process model discovery technique that leverages hierarchical information in event logs, typically available for software system cases.* This technique allows us to analyze the operational processes of software systems under real-life conditions at multiple levels of granularity. An implementation of the proposed algorithm is made available via the Statechart plugin in the process mining framework ProM [30]. The Statechart workbench provides an intuitive way to discover, explore and analyze hierarchical behavior, integrates with existing ProM plugins and links back to the source code in Eclipse. Our experimental results, based on real-life (software) event logs, demonstrate the feasibility and usefulness of the approach and *show the huge potential to speed up discovery by exploiting the available hierarchy.*
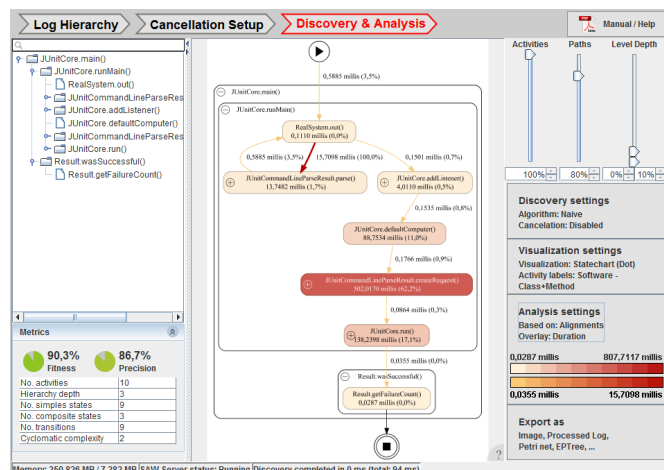
Future work aims to uncover exceptional and error control flows (i.e., try-catch and cancellation patterns), provide reliability analysis, and better support multi-threaded and distributed software. In addition, enabling the proposed techniques in a streaming context could provide valuable real-time insight into software in its natural environment. Furthermore, since (software) event logs can be very large, using a streaming context means we do not have to use a large amount of storage.

REFERENCES

[1] C. Ackermann, M. Lindvall, and R. Cleaveland. Recovering views of inter-system interaction behaviors. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 53–61. IEEE, Oct 2009.

[2] A. Adriansyah. *Aligning observed and modeled behavior*. PhD thesis, Technische Universiteit Eindhoven, 2014.

[3] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Automated reverse engineering of UML sequence diagrams for dynamic web applications. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 287–294. IEEE, April 2009.

[4] A. Amighi, P. de C. Gomes, D. Gurov, and M. Huisman. Sound control-flow graph extraction for Java programs with exceptions. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 33–47, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Apache Commons Documentation Team. Apache Commons Crypto 1.0.0-src. http://commons.apache.org/proper/commons-crypto. [Online, accessed 3 March 2017].

[6] I. Beschastnikh. Synoptic Model Inference. https://github.com/ModelInference/synoptic. [Online, accessed 31 August 2017].

[7] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst. Synoptic: Studying logged behavior with inferred models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 448–451, New York, NY, USA, 2011. ACM.

[8] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, New York, NY, USA, 2014. ACM.

[9] S. Birkner, E. Gamma, and K. Beck. JUnit 4 - Getting Started. https://github.com/junit-team/junit4/wiki/Getting-started. [Online, accessed 19 July 2016].

[10] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *Software Engineering, IEEE Transactions on*, 32(9):642–663, Sept 2006.

[11] L. C. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. *2013 20th Working Conference on Reverse Engineering (WCRE)*, page 57, 2003.

[12] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In *On the Move to Meaningful Internet Systems: OTM 2012*, volume 7565 of *Lecture Notes in Computer Science*, pages 305–322. Springer Berlin Heidelberg, 2012.

[13] S. Chiba. Javassist – a reflection-based programming wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, page 5, October 1998.

[14] S. Chiba. Load-time structural reflection in Java. In E. Bertino, editor, *European Conference on Object-Oriented Programming 2000 – Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Berlin Heidelberg, 2000.

[15] R. Conforti, M. Dumas, L. García-Bañuelos, and M. La Rosa. BPMN miner: Automated discovery of BPMN process models with hierarchical structure. *Information Systems*, 56:284 – 303, 2016.

[16] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190 – 210, 2006. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA '04).

[17] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution Patterns in Object-oriented Visualization. *Proc. COOTS*, pages 219–234, 1998.

[18] E. Gamma and K. Beck. JUnit 4.12. https://mvnrepository.com/artifact/junit/junit/4.12. [Online, accessed 19 July 2016].

[19] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ. Aspect-Oriented Programming in Java*, volume 456. John Wiley & Sons, 2003.

[20] S. L. Graham, P. B. Kessler, and M. K. Mckusick. gprof: call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction - SIGPLAN '82*, pages 120–126, New York, New York, USA, 1982. ACM Press.

[21] C. W. Günther and W. M. P. van der Aalst. Fuzzy mining – adaptive process simplification based on multi-perspective metrics. In *Business Process Management*, pages 328–343. Springer, 2007.

[22] C. W. Günther and H. M. W. Verbeek. XES – standard definition. Technical Report BPM reports 1409, Eindhoven University of Technology, 2014.

[23] M. J. H. Heule and S. Verwer. *Exact DFA Identification Using SAT Solvers*, pages 66–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[24] R. P. Jagadeesh Chandra Bose and W. M. P. van der Aalst. *Abstractions in Process Mining: A Taxonomy of Patterns*, volume 5701 of *Lecture Notes in Computer Science*, pages 159–175. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[25] R. P. Jagadeesh Chandra Bose, E. H. M. W. Verbeek, and W. M. P. van der Aalst. *Discovering Hierarchical Process Models Using ProM*, volume 107 of *Lecture Notes in Business Information Processing*, pages 33–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[26] I. Jonyer, L. B. Holder, and D. J. Cook. MDL-Based Context-Free Graph Grammar Induction. *International Journal on Artificial Intelligence Tools*, 13(01):65–79, mar 2004.

[27] R. Kollmann and M. Gogolla. Capturing dynamic program behaviour with UML collaboration diagrams. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 58–67, 2001.

[28] E. Korshunova, M. Petkovic, M. G. J. van den Brand, and M. R. Mousavi. CPP2XMI: Reverse engineering of UML class, sequence, and activity diagrams from C++ source code. In *2006 13th Working Conference on Reverse Engineering*, pages 297–298, Oct 2006.

[29] Y. Labiche, B. Kolbah, and H. Mehrfard. Combining static and dynamic analyses to reverse-engineer scenario diagrams. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 130–139. IEEE, Sept 2013.

[30] M. Leemans. Statechart plugin for ProM 6. https://svn.win.tue.nl/repos/prom/Packages/Statechart/. [Online, accessed 15 July 2016].

[31] M. Leemans. JUnit 4.12 software event log. http://doi.org/10.4121/uuid:cfed8007-91c8-4b12-98d8-f233e5cd25bb, 2016.

[32] M. Leemans. Apache Commons Crypto 1.0.0 - Stream Cbc-Nopad unit test software event log. http://doi.org/10.4121/uuid:bb3286d6-dde1-4e74-9a64-fd4e32f10677, 2017.

[33] M. Leemans. NASA Crew Exploration Vehicle (CEV) software event log. http://doi.org/10.4121/uuid:60383406-ffcd-441f-aa5e-4ec763426b76, 2017.

[34] M. Leemans and W. M. P. van der Aalst. Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 44–53, Sept 2015.

[35] S. J. J. Leemans. *Robust process mining with guarantees*. PhD thesis, Eindhoven University of Technology, May 2017.

[36] 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 J. M. Colom and J. Desel, editors, *Application and Theory of Petri Nets and Concurrency*, pages 311–329. Springer Berlin Heidelberg, 2013.

[37] NASA. JPF Statechart and CEV example. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-statechart. [Online, accessed 3 March 2017].

[38] C. G. Nevill-Manning and I. H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, aug 1997.

[39] R. Oechsle and T. Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 176–190. Springer Berlin Heidelberg, 2002.

[40] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 254–263, New York, NY, USA, 2005. ACM.

[41] P. Siyari, B. Dilkina, and C. Dovrolis. Lexis: An Optimization Framework for Discovering the Hierarchical Structure of Sequential Data. *Gecco*, pages 421–434, feb 2016.

[42] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, CRPIT '02, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[43] W. Steeman. BPI Challenge 2013, incidents. http://dx.doi.org/10.4121/uuid:500573e6-accc-4b0c-9576-aa5468b10cee, 2013.

[44] T. Systä, K. Koskimies, and H. Müller. Shimba – an environment for reverse engineering Java software systems. *Software: Practice and Experience*, 31(4):371–394, 2001.

[45] P. Tonella and A. Potrich. Reverse engineering of the interaction diagrams from C++ code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 159–168, Sept 2003.

[46] W. M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer-Verlag, Berlin, 2016.

[47] W. M. P. van der Aalst, A. K. Alves de Medeiros, and A. J. M. M. Weijters. Genetic Process Mining. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 48–69. Springer-Verlag, Berlin, 2005.

[48] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, 9(1):87–111, 2010.

[49] 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*, 16(9):1128–1142, 2004.

[50] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik. *Process Discovery Using Integer Linear Programming*, pages 368–387. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[51] B.F. van Dongen. BPI Challenge 2012. http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f, 2012.

[52] S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. *Avoiding Over-Fitting in ILP-Based Process Discovery*, pages 163–171. Springer International Publishing, Cham, 2015.

[53] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. XES, XESame, and ProM 6. In P. Soffer and E. Proper, editors, *Information Systems Evolution*, volume 72 of *Lecture Notes in Business Information Processing*, pages 60–75. Springer Berlin Heidelberg, 2011.

[54] N. Walkinshaw. EFSMInferenceTool. https://bitbucket.org/nwalkinshaw/efsminferencetool. [Online, accessed 19 July 2016].

[55] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

[56] A. J. M. M. Weijters and J. T. S. Ribeiro. Flexible Heuristics Miner (FHM). In *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 310–317, April 2011.

[57] Z. Yao, Q. Zheng, and G. Chen. AOP++: A generic aspect-oriented programming framework in C++. In R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 94–108. Springer Berlin Heidelberg, 2005.