Component Behavior Discovery from Software Execution Data

Cong Liu, Boudewijn van Dongen, Nour Assy and Wil M.P. van der Aalst
Department of Mathematics and Computer Science,
Eindhoven University of Technology, Eindhoven, The Netherlands.
E-mail: {c.liu.3, B.F.v.Dongen, n.assy, w.m.p.v.d.aalst}@tue.nl

***Abstract*—Tremendous amounts of data can be recorded during software execution. This provides valuable information on software runtime analysis. Many crashes and exceptions may occur, and it is a real challenge to understand how software is behaving. Software is usually composed of various *components*. A component is a nearly independent part of software that fullfills a clear function. *Process mining* aims to discover, monitor and improve real processes by extracting knowledge from event logs. This paper presents an approach to utilize process mining as a tool to discover the real behavior of software and analyze it. The unstructured software execution data may be too complex, involving multiple interleaved components, etc. Applying existing process mining techniques results in spaghetti-like models with no clear structure and no valuable information that can be easily understood by end. In this paper, we start with the observation that software is composed of components and we use this information to decompose the problem into smaller independent ones by discovering a behavioral model per component. Through experimental analysis, we illustrate that the proposed approach facilitates the discovery of more understandable software models. All proposed approaches have been implemented in the open-source process mining toolkit ProM.**

I. Introduction

During the execution of software, execution data can be recorded. By exploiting the recorded data, one can discover behavioral models to describe the actual execution of software. The discovered models can provide insight regarding real usage of software, motivate novel idea on model-based testing, enable software usability improvements, localize performance problems and architectural challenges, etc. *Process mining* [15] aims at extracting process models from event logs. There are three major types of process mining: (1) process discovery; (2) conformance checking; and (3) enhancement. *Process Discovery* takes an event log as input and produces a process model. *Conformance Checking* aims to compare process models discovered or hand-made with real-behavior recorded in event logs. *Enhancement* also takes an event log and an existing model as inputs and tries to improve or extend the model using additional information in the log. Enhancement can be used to highlight bottlenecks, predict performance, etc. With the development of process mining techniques on the one hand, and the growing availability of software execution data on the other hand, a new form of software analytics comes into reach, i.e., applying process mining techniques to analyze software execution data. This inter-disciplinary research area is called *Software Process Mining (SPM)* [14].

The software execution data consist of method calls and object information. Usually, each software run may last for a long time and involve multiple interleaved component executions. Applying existing process mining techniques typically results in flat and spaghetti-like models with no clear structure and no valuable information for comprehension. Given the observation that software typically involve a set of components, we propose to use this component information to decompose the problem into smaller independent ones by discovering a behavioral model per component. Starting from the original execution data, we first propose a novel idea that allows us to identify component instance (one independent run of a software component) as the $case$ notion of software event log. As process mining algorithms expect an event log as input, the choice of *case* notion determines the scope of the discovered models [1]. Besides components, software usually have a hierarchical structure represented as multi-level nested method calls, hence the discovered behavioral models should also depict this hierarchy nature. After obtaining a software event log for each component, we then recursively transform the log to a hierarchical one using the calling relation among methods. Given a hierarchical software event log, we use existing techniques to discover a hierarchical process model. Existing approaches are unable to discover software (component) behavioral models with hierarchies. To the best of our knowledge this is the first paper to discover such models.

The rest of this paper is organized as follows. Section II presents a brief review of related work. Section III defines some preliminaries. Section IV presents the software component behavior discovery approach. Section VI presents experimental results. Section V introduces ProM plugin implementation. Finally, Section VII concludes the paper.

II. Related Work

A. Software Dynamic Analysis

Software dynamic analysis is used to understand the behavior of software by exploiting its execution data. Several techniques and tools have been presented to extract information from running software. Most existing approaches, such as [9] and [20], generate automaton-based models using different variants of the *K-Tail* algorithm which was first defined by *Biermann* and *Feldman* [2]. However, these techniques cannot discover concurrency explicitly, resulting in a so-called state explosion for complex models. Although automaton-based

models are popular in this area, there are several techniques to learn other types of models. For example, some techniques visualize software execution traces as sequence diagrams [11] and some of them are extended with loops [3]. Similarly, the sequence diagram-based models also lack concurrency description. Moreover, each sequence diagram or automaton-based model only describes the behavior of a single execution trace. Given software execution data referring to multiple traces, these existing approaches will obtain an excessive number of behavioral models rather than a compact model for the whole data. In addition, considering the hierarchy nature of a software, the discovered flat sequence diagrams or flat automation-based models cannot accurately capture the real behavior in a meaningful way.

B. Process Mining

Process mining deals with discovering, monitoring and improving business processes by extracting knowledge from event logs [15]. The α-Algorithm [16] defines four kinds of ordering relations (directly-follow, causality, choice and concurrency), based on which it constructs a workflow net. More recently, a family of inductive process mining techniques [7] using process trees are proposed to handle incomplete and noisy event logs. Different from these process discovery algorithms that produce flat models, a two-phase mining approach using pattern detection techniques was introduced by *Li et al.* [8] to discover hierarchical process models. Two types of patterns, loops and maximal repeats, are considered. Once patterns are identified, all their occurrences are replaced by an abstract activity and extracted as sub-process logs. Different from them, we propose to detect software-related patterns and discover hierarchical models organized in calling relation. Conforti *et al.* [4] present a technique to discover *BPMN* model with sub-processes, multi-instance markers, etc. It relies on some special attributes (primary and foreign keys) to infer dependencies between parent and sub-processes and multi-instance markers. Another research area is artifact-centric process discovery, which aims to discover artifact lifecycle behavioral models and interactions between them [10]. An artifact describes the lifecycle of a business object (e.g. a purchase order). Our work goes into similar direction and aims to discover software behavioral models.

C. Software Process Mining

Software process mining enables the extraction of knowledge from software execution data, which helps software analysts better understand software behavior. Our paper belongs to the software runtime behavior discovery spectrum. One of the first papers addressing this problem using process mining is [17]. For the mining of software systems, the recorded events explicitly refer to parts of the system (components, services, etc.). References to system parts facilitate the generation of localized event logs. A generic process discovery approach is proposed based on localized event logs. Experimental results show that location information indeed helps to improve the quality of the discovered models. *Leemans and van der Aalst* [6] analyze the operational processes of software systems, and process mining techniques are applied to obtain precise and formal models using real-life event logs. They propose to discover flat behavior models using Inductive Miner [7]. However, the hierarchical structure of software is not fully considered. More recently, *van der Aalst* [1] propose to analyze software systems under real-life circumstances using process mining, i.e., the "*Big Software on the Run* "project [1]. Our current work is also in the context of this project.

III. Preliminaries

Let S be a set and we use $\emptyset$ for the empty set. We use the standard $|S|$ to denote the number of elements in set S. The powerset of S is denoted by $\mathcal{P}(S) = \{S' | S' \subseteq S\}$. $f \in X \rightarrow Y$ is a function, i.e., $dom(f)$ is the domain and $rng(f) = \{f(x) | x \in dom(f)\}$ is the range. A *multi-set (or bag)* over S is a set where elements can appear multiple times, e.g., $m = [p^3, q^2]$ is a multi-set over $S = \{p, q\}$ where $m(p) = 3$, $m(q) = 2$. The set of all multi-sets over S is denoted by $\mathbb{N}^S$. We use $+$ and $-$ for the union and difference of two multi-sets. Both of them are defined the same way as set.

A sequence over S of length n is a function $\sigma : \{1, 2, ..., n\} \rightarrow S$. If $\sigma(1) = a_1, \sigma(2) = a_2, ...\sigma(n) = a_n$, we write $\sigma = \langle a_1, a_2, ...a_n \rangle$. A sequence of length 0 is called the empty sequence, denoted by $\langle\rangle$. The length of a sequence σ is denoted by $|\sigma|$, e.g., $|\langle\rangle| = 0$. The set of all finite sequences over set S is denoted by S^*. Let $u, v \in S^*$ be two sequences, the *concatenation* operation denoted by $\sigma = u \circ v$ is defined as σ: $\{1, 2, ...|u| + |v|\} \rightarrow S$, such that $\sigma(i) = u(i)$ for $1 \leq i \leq |u|$, and $\sigma(i) = v(i - |u|)$ for $|u| + 1 \leq i \leq |u| + |v|$.

Definition 1: (**Sequence Projection**) Let X be a set and $Q \subseteq X$ be its subset. $\restriction_Q \in X^* \rightarrow Q^*$ is a projection function and is defined recursively: $\langle\rangle\restriction_Q = \langle\rangle$; and for $\sigma \in X^*$ and $x \in X$:

$$(\langle x \rangle \circ \sigma) \restriction_Q = \begin{cases} \sigma\restriction_Q & if \quad x \notin Q \\ \langle x \rangle \circ (\sigma\restriction_Q) & if \quad x \in Q \end{cases} \tag{1}$$

Definition 2: (**Labeled Petri net** [13]) A *Labeled Petri net* is a 4-tuple $PN = (P, T, F, l)$, satisfying (1) $P \cap T = \emptyset$, $P \cup T \neq \emptyset$ where P is the place set and T is the transition set; (2) $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation; and (3) $l : T \rightarrow \mathscr{A}$ is a labeling function where $\mathscr{A}$ is a set of labels and $\tau \in \mathscr{A}$ denotes invisible label.

Given a $PN = (P, T, F, l)$, we define the preset and post-set of transitions and places. For each $x \in P \cup T$, the set ${}^\bullet x = \{y | (y, x) \in F\}$ is the preset (input) of x and $x^\bullet = \{y | (x, y) \in F\}$ is the postset (output) of x. To describe the semantics of a labeled Petri net, we use *markings*. A marking m of PN is a multiset of places, i.e., $m \in \mathbb{N}^P$, indicating how many tokens each place contains. Markings are state of a net. (PN, m_0) is a *marked net* where m_0 is its *initial marking*. A transition $t \in T$ is *enabled* in marking $m \in \mathbb{N}^P$, denoted as $(PN, m)[t >$ if and only if $\forall p \in {}^\bullet t : m(p) \geq 1$. An enabled transition t may *fire* and results in a new marking m' with $m' = m -^\bullet t + t^\bullet$, denoted by $(PN, m)[t > (PN, m')$.

[1] http://www.3tu-bsr.nl/doku.php?id=start

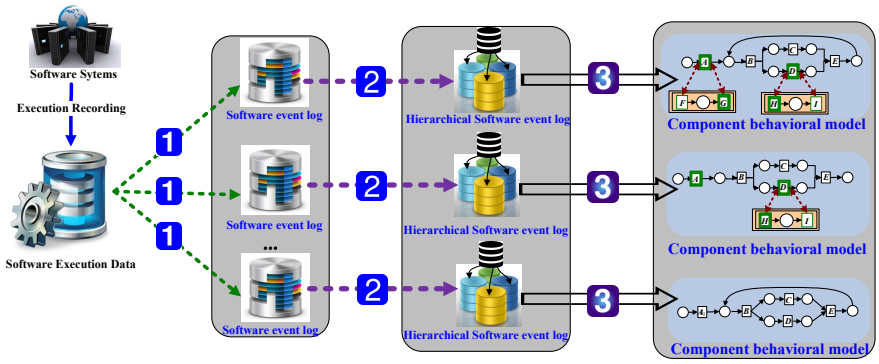


Fig. 1. Software Component Behavior Discovery: An Overview of the Approach

IV. Component Behavior Discovery

This section details the main approach to discover component behavioral models from software execution data. In *Section* IV-A, we give an overview of our approach. In *Sections* IV-B-IV-C, we present our main discovery approach.

A. Approach Overview

The input of our approach is software execution data, which can be obtained by instrumenting and monitoring software execution. Fig. 1 shows an overview of the approach which consists of three main steps:

1. Component Instance Identification. Software typically contains a set of *components*. A component is a non-trivial, nearly independent and replaceable part of software that fullfills a clear function in the context of a well-defined architecture [5]. Starting from the original software execution data, we first propose a novel idea to identify component instances. These serve as the basic *case* notion to generate a software event log for each component. Here, a component instance refers to one independent run of a software component.

2. Hierarchical Software Event Log Construction. Because a software (component) usually has a hierarchical structure represented as multi-level nested method calls, the discovered behavioral model should depict this hierarchy nature. For each component, we recursively transform its software event log to a hierarchical one using calling relations among methods.

3. Component Behavioral Model Discovery. For each component, we discover a hierarchical behavioral model from its corresponding hierarchial software event log. Given the hierarchy of a software event log, we only need to traverse through different levels of the log and discover a process model for each sub-log. Note that we can use any existing process discovery approach in this step.

B. Software Component Instance Identification

The starting point of our component behavior discovery approach is the software execution data and the *method call* is the basic unit in these data.

Definition 3: (**Method Call, Attribute**) Let $\mathscr{U}_M$ be the method call universe. A method call may have various attributes. Let $\mathscr{U}_A$ be the attribute universe, i.e., a set of attribute names. For each method call $m \in \mathscr{U}_M$ and attribute $AN \in \mathscr{U}_A$, $\#_{AN}(m)$ is the value of AN for m.

In the following, let $\mathscr{U}_N$ be the method universe, $\mathscr{U}_{CL}$ be the class universe, $\mathscr{U}_{CO}$ be the class object universe, and $\mathscr{U}_T$ be the time universe. We assume that each method call $m \in \mathscr{U}_M$ has at least the following standard (mandatory) attributes:

- $\#_{calleeM}(m) \in \mathscr{U}_N$ is the callee method name of m;
- $\#_{calleeC}(m) \in \mathscr{U}_{CL}$ is the callee class name of m;
- $\#_{calleeCO}(m) \in \mathscr{U}_{CO}$ is callee class object identifier of m;
- $\#_{callerM}(m) \in \mathscr{U}_N$ is the caller method name of m;
- $\#_{callerC}(m) \in \mathscr{U}_{CL}$ is the caller class name of m;
- $\#_{callerCO}(m) \in \mathscr{U}_{CO}$ is caller class object identifier of m;
- $\#_{StartTime}(m) \in \mathscr{U}_T$ is the start timestamp of m; and
- $\#_{EndTime}(m) \in \mathscr{U}_T$ is the end timestamp of m.

Definition 4: (**Software Execution Trace/Data**) Let $\mathscr{U}_M$ be the method call universe. $\sigma \subseteq \mathscr{U}_M$ is a software execution trace. $SD \subseteq \mathcal{P}(\mathscr{U}_M)$ is software execution data such that $\forall \sigma_i, \sigma_j \in SD : \sigma_i \cap \sigma_j = \emptyset \vee \sigma_i = \sigma_j$.

Given a software system, it contains a set of interacting components. According to Definition 4, software execution data are composed of a set of execution traces, each describing a set of interacting components. The term of component is generic, and typically consists of a group of classes. Given software whose development documents are well organized and kept, we have information of how classes are grouped to form components [5]. Therefore, we can pre-process software execution data to obtain a set of software component execution data. Consider for example the software execution data in Table I, it is composed of one software execution trace σ_e. Each row corresponds to a method call, we use *ID* to uniquely represent each method call, e.g., e_1 is the first method call.

A component $C \subseteq \mathscr{U}_{CL}$ is a set of classes and $\mathcal{P}(\mathscr{U}_{CL})$ is the component universe. In this paper, we assume classes are uniquely identifiable by using packages as the prefix. Given a piece of software, $COM \subseteq \mathcal{P}(\mathscr{U}_{CL})$ is its component set. We assume components of the same software cannot overlap, i.e., for any $C_i, C_j \in COM$, $C_i \cap C_j = \emptyset$ or $C_i = C_j$. For instance, we assume the software execution data in Table I has two components: $C_1 = \{class1, class2\}$ and $C_2 = \{MainClass\}$.

TABLE I
AN EXAMPLE OF SOFTWARE EXECUTION DATA

ID	Callee Method	Callee Class	Callee Class Object	Caller Method	Caller Class	Caller Class Object
e_1	init()	class2	@5746e7cc	main()	MainClass	@mainclass
e_2	init()	class1	@3b7359cb	main()	MainClass	@mainclass
e_3	setClass1()	class2	@5746e7cc	init()	Class1	@3b7359cb
e_4	perform()	class1	@3b7359cb	main()	MainClass	@mainclass
e_5	work()	class2	@5746e7cc	perform()	Class1	@3b7359cb
e_6	init()	class1	@614b152d	main()	MainClass	@mainclass
e_7	setClass1()	class2	@5746e7cc	init()	Class1	@614b152d
e_8	perform()	class1	@614b152d	main()	MainClass	@mainclass
e_9	work()	class2	@5746e7cc	perform()	Class1	@614b152d
e_{10}	main()	MainClass	@mainclass	null	null	null

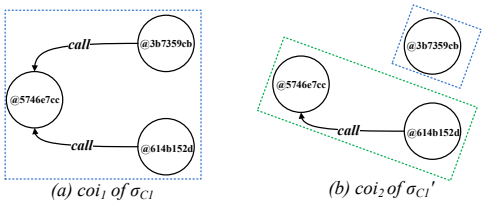
Definition 5: (**Software Component Execution Trace/-Data**) Let $COM \subseteq \mathcal{P}(\mathscr{U}_{CL})$ be the component set of a piece of software, $SD \subseteq \mathcal{P}(\mathscr{U}_M)$ be its software execution data and $C \in COM$ be one software component. For each $\sigma \in SD$, $\sigma_C = \{e \in \sigma | \#_{calleeC}(e) \in C\}$ is the software execution trace of component C. $SD_C = \{\sigma_C | \sigma \in SD\}$ is the software execution data of component C.

According to Definition 5, the example execution traces of C_1 and C_2 in Table I are $\sigma_{C1} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$ and $\sigma_{C2} = \{e_{10}\}$.

A component instance represents one run of the component and a software component execution trace may contain one or more interleaved component instances. In this work, we propose to split each component execution trace into different independent ones. To identify a component instance, we make the observation that for an object-oriented software, a component instance starts when its class objects are constructed. Then a component instance contains: (1) the first class object constructed by an external component; and (2) all class objects that interact (directly or indirectly) with the first constructed object. To identify independent component instances, we propose to: (1) construct a *Class Object Interaction* graph for each software component execution trace; and (2) detect its *weakly connected components* which represent different instances.

Definition 6: (**Class Object Interaction Graph**) Let $C \in COM$ be one software component and SD_C be its software component execution data. For each $\sigma_C \in SD_C$, $G_{\sigma_C} = (V_{\sigma_C}, R_{\sigma_C})$ is the *Class Object Interaction* (*COI*) graph of σ_C such that (1) $V_{\sigma_C} = \{o \in \mathscr{U}_{CO} | \exists e \in \sigma_C : \#_{calleeCO}(e) = o\}$ and (2) $R_{\sigma_C} = \{(o_i, o_j) \in V_{\sigma_C} \times V_{\sigma_C} | \exists e \in \sigma_C : \#_{callerCO}(e) = o_i \wedge \#_{calleeCO}(e) = o_j\}$.

According to Definition 6, a *COI* graph contains (1) a set of class objects, i.e. vertices; and (2) a set of interaction relations between them, i.e., edges. It is worth noting that the interaction relation among class objects is obtained from the calling relation among method calls. Given a class object interaction graph *coi*, we use *coi.V* and *coi.R* to represent its vertex and edge sets. Considering the example trace σ_{C1} of C_1, its class object interaction graph, denoted as coi_1, is shown in Fig. 2*(a)*. Its class object set coi_1*.V*={*@5746e7cc, @3b7359cb, @614b152d*} and interaction relation set coi_1*.R*={*(@3b7359cb, @5746e7cc), (@614b152d, @5746e7cc)*}. If we modify σ_{C1} to $\sigma'_{C1} = \{e_1, e_2, e_4, e_6, e_7, e_8, e_9\}$, its class object interaction graph, denoted as coi_2, is shown in Fig. 2*(b)*. The class object set is coi_2*.R*={*@5746e7cc, @3b7359cb, @614b152d*} and the interaction relation set is coi_2*.V*={*(@614b152d, @5746e7cc)*}.



(a) coi_1 of σ_{C1} (b) coi_2 of σ_{C1}'

Fig. 2. Examples of Class Object Interaction Graphs for two Different Traces

After constructing a class object interaction graph, the set of vertices (class objects) included in each *weakly connected component* corresponds to a component instance. A *weakly connected component* of a directed graph is a maximal group of vertices that are mutually reachable by violating the edge directions. It can be easily obtained by several iterative *DFS-traverse*s [12] of its corresponding undirected graph. Consider the execution trace σ_{C1} and its *COI* graph in Fig. 2*(a)*, it has one weakly connected component, which means this trace only relates to one component instance. Consider the execution trace σ'_{C1} and its *COI* graph in Fig. 2*(b)*, it has two weakly connected components, which means this trace relates to two component instances and will be split into two different ones.

Definition 7: (**Software Component Instance**) Let $\sigma_C \in SD_C$ be a software component execution trace and $G_{\sigma_C} = (V_{\sigma_C}, R_{\sigma_C})$ be its corresponding class object interaction graph. $I_{\sigma_C} = \{I^i_{\sigma_C} | i \geq 1\} \subseteq \mathcal{P}(V_{\sigma_C})$ is the component instance set of σ_C such that (1) $\bigcup_{i \geq 1} I^i_{\sigma_C} = V_{\sigma_C}$; (2) $\forall_{1 \leq i < j} I^i_{\sigma_C} \cap I^j_{\sigma_C} = \emptyset \wedge R_{\sigma_C} \cap ((I^i_{\sigma_C} \times I^j_{\sigma_C}) \cup (I^j_{\sigma_C} \times I^i_{\sigma_C})) = \emptyset$; and (3) $\nexists I'_{\sigma_C} \subseteq \mathcal{P}(V_{\sigma_C}) : |I_{\sigma'_C}| > |I_{\sigma_C}|$.

By recursively applying Definition 7 to each software execution trace of a component, we can identify all instances of this component. To enable the discovery of the behavior of software components using process mining techniques, software event logs of each component are required as input.

A software event log is obtained from a software component execution data by instance identification. Each software case

is defined as a sequence of software events refer to one identified component instance, and a software event refers to a method call. Therefore, all attributes defined on method calls are included for software events. To explicitly capture all information that may exist in software event logs, we formalize software events and their attributes as follows:

Definition 8: (**Software Event**) Let SD_C be the software execution data of component C. $E = \bigcup_{\sigma_C \in SD_C} \{e | e \in \sigma_C\}$ is the software event set of component C. All events $e \in E$ have the following standard attributes: (1) $\#_{case}(e) = I^i_{\sigma_C}$ where $I^i_{\sigma_C} \in I_{\sigma_C}$ and $\#_{calleeCO}(e) \in I^i_{\sigma_C}$, is the *case* of e; (2) $\#_{Act}(e) = (\#_{calleeC}(e), \#_{calleeM}(e))$ is the *activity* of e; and (3) $\#_{time}(e) = \#_{StartTime}(e)$ is the *timestamp* of e.

A software event is identified by its activity name, e.g., $\#_{Act}(e_1) = (class2, init())$, and is denoted as $class2.init()$.

Definition 9: (**Software Case, Software Event Log**) Let E be the software event set of component C. The software event log over E, denoted as $L = \{\varphi | \varphi \in E^*\}$, is a finite set of cases such that (1) each software event appears only once in each case, i.e., $\forall \varphi \in L,\ 1 \leqslant i < j \leqslant |\varphi| :\ \varphi_i \neq \varphi_j$; (2) each software event appears in one and only one case, i.e., $\forall e \in E, \exists \varphi \in L : e \in \varphi\ \wedge\ \nexists \varphi' \in L, \varphi' \neq \varphi : e \in \varphi'$; (3) all events in the same case have the same $\#_{case}$ attribute value, i.e., $\forall \varphi \in L, 1 \leqslant i, j \leqslant |\varphi| : \#_{case}(\varphi_i) = \#_{case}(\varphi_j)$; (4) all events with the same $\#_{case}$ attribute value are in the same case, i.e., $\forall e_i, e_j \in E : \#_{case}(e_i) = \#_{case}(e_j) \Rightarrow \exists \varphi \in L : e_i, e_j \in \varphi$; and (5) all events of the same case are ordered by the timestamp, i.e., $\forall \varphi \in L, 1 \leq i < j \leq |\varphi| : \#_{time}(\varphi_i) \leq \#_{time}(\varphi_j)$.

In the following discussion, we use $E(L) = \bigcup_{\varphi \in L} \{e | e \in \varphi\}$ to denote the event set of a software event log L.

C. Component Behavioral Model Discovery

The software event log of a component contains a set of cases, each describing one component instance, based on which we can discover the component behavioral model. Because a component usually has a hierarchical structure represented as multi-level nested method calls, the discovered component behavioral model should be able to depict this. After obtaining a software event log, we recursively transform it to a hierarchical one using the calling relation among methods. To define the hierarchical structure, we first define its root that we call main (or top-level) event log. It is composed of a set of events whose caller classes do not belong to the current component, i.e., they are invoked by other components.

Definition 10: (**Main Event Log**) Let $C \in \mathcal{P}(\mathscr{U}_{CL})$ be a software component, L be its software event log and $E(L)$ be its event set. $ME(L) = \{e \in E(L) | \exists \varphi \in L : e \in \varphi\ \wedge\ \#_{calleeC}(e) \in C \wedge \#_{callerC}(e) \notin C\}$ is the main event set of L. $mLog_0 = \{\varphi\restriction_{ME(L)} | \varphi \in L\}$ is the main event log of L.

To introduce the nesting relationships in the hierarchical structure, we define the nested event set and events invoked by each nested event in the following.

Definition 11: (**Nested Event Set**) Let L be a software event log and $E(L)$ be its event set. $NE(L) = \{ne \in E(L) | \exists \varphi \in L, e \in \varphi : ne \in \varphi \wedge \#_{callerCO}(e) = \#_{calleeCO}(ne)\}$ is the nested event set of L.

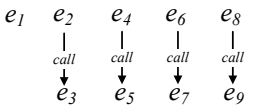


Fig. 3. Hierarchical Structure of σ_{C1}

Definition 12: (**Invoked Event Set**) Let L be a software event log, $E(L)$ be its event set and $NE(L)$ be its nested event set. The event set of L invoked by $ne \in NE(L)$ is defined as: $IE(ne, L) = \{e \in E(L) | \exists \varphi \in L : e \in \varphi \wedge \#_{callerCO}(e) = \#_{calleeCO}(ne)\}$.

According to Definitions 11-12, the nested event set of σ_{C1} is $\{e_2, e_4, e_6, e_8\}$ and the invoked event set of e_2 is $\{e_3\}$.

Given a software event log L and $mLog_0$ is its main event log, $NE_H(mLog_0) = E(mLog_0) \cap NE(L)$ is the nested event set of $mLog_0$ and $NE^C_H(mLog_0) = \{\#_{Act}(e) | e \in NE_H(mLog_0)\}$ is the nested event class set of $mLog_0$.

Definition 13: (**Hierarchical Software Event Log**) Let L be a software event log. $\mathcal{HL}(L) = (mLog_0, HL(mLog_0))$ is defined as the hierarchical software event log of L where: $mLog_0$ is the main event log of L; and

- $HL(mLog_0) = \emptyset$ if $NE_H(mLog_0) = \emptyset$; otherwise
- $HL(mLog_0) = \{(nec, ILog_{nec}, HL(ILog_{nec})) | nec \in NE^C_H(mLog_0)\}$ where $ILog_{nec} = \bigcup_{ne \in NE_H(mLog_0)} \{\varphi\restriction_{IE(ne,L)} | \varphi \in L \wedge \#_{Act}(ne) = nec\}$ is the invoked event log of L by nec.

Taking a software event log with only trace σ_{C1} as example. Its hierarchical structure is shown in Fig. 3. According to Definitions 10-11, its main event log is $mLog_0 = [\langle e_1, e_2, e_4, e_6, e_8\rangle]$, its nested event set is $NE_H(mLog_0) = \{e_2, e_4, e_6, e_8\}$ and nested event class set is $NE^C_H(mLog_0) = \{class1.init(), class1.perform()\}$. Then, the event logs invoked by $class1.init()$ and $class1.perform()$ are $ILog_{class1.init()} = [\langle e_3\rangle, \langle e_7\rangle]$ and $ILog_{class1.perform()} = [\langle e_5\rangle, \langle e_9\rangle]$. The recursive definition stops at this level as the invoked event logs of $class1.init()$ and $class1.perform()$ do not contain any nested events.

Given the hierarchy of a software event log, we need to discover a hierarchical process model to explicitly show the behavior. For this step, we can recursively apply any process discovery approach, including techniques such as the *Inductive Miner*[7]. Instead of obtaining a normal Petri net, the discovered Petri net is extended with a set of nested transitions.

Definition 14: (**Petri net with Nested Transitions**) A *Petri net with nested transitions* is a 2-tuple $PN_N = (PN, \mathcal{N})$ such that (1) $PN = (P, T, F, l)$ is a labeled Petri net; and (2) $\mathcal{N} : T \to \{A, N\}$ is a mapping function such that $\forall t \in T, \mathcal{N}(t) = A$ represents t is an atomic transition and $\mathcal{N}(t) = N$ represents t is a nested transition.

Given a PN_N, we denote by $T_a = \{t \in T | \mathcal{N}(t) = A\}$ the atomic transition set and $T_n = \{t \in T | \mathcal{N}(t) = N\}$ the nested transition set. In the following, we will use the notation T_{a0} and T_{n0} for PN_{N0}. Fig. 4 shows a simple example of

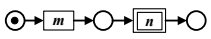


Fig. 4. An Example of Petri Net with Nested Transitions

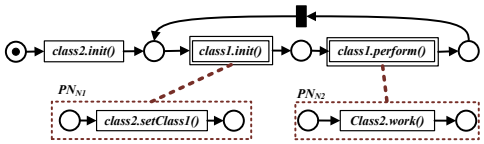


Fig. 5. An Example Hierarchical Petri Net with Nested Transitions

Petri net with nested transitions where an atomic transition is drawn using single-line rectangle (*m*) and a double-line rectangle (*n*) is used to draw a nested transition.

Let $\mathscr{U}_{PN}$ be the universe of Petri net with nested transitions. Based on calling relation, we recursively define the *hierarchical Petri net with nested transitions*.

Definition 15: (**Hierarchical Petri net with Nested Transitions**) $\mathcal{HPN} = (PN_{N0}, HPN(PN_{N0}))$ is defined as a *hierarchical Petri net with nested transitions* where: $PN_{N0} \in \mathscr{U}_{PN}$ is the top-level Petri net with nested transitions; and

- $HPN(PN_{N0}) = \emptyset$ if $T_{n0} = \emptyset$; otherwise
- $HPN(PN_{N0}) = \{(t_i, PN_{Ni}, HPN(PN_{Ni}))|t_i \in T_{n0}\}$ where $PN_{Ni} \in \mathscr{U}_{PN}$ is the Petri net with nested transitions that is called by t_i.

The behavior discovery approach takes a hierarchical software event log as input, and discovers a hierarchical Petri net with nested transitions. As the hierarchical software event log already has the notion of hierarchies, we only need to (1) traverse different levels of software event log to discover its PN_N and (2) construct the mapping from each nested transition (nested event class) to its corresponding PN_N.

Definition 16: (**Discovery**) Let $\mathscr{U}_{HL}$ be the universe of hierarchical software event log and $\mathscr{U}_{HPN}$ be the universe of hierarchical Petri net with nested transitions. $\mathcal{D} : \mathscr{U}_{HL} \rightarrow \mathscr{U}_{HPN}$ is a discovery function such that for each $\mathcal{HL}(L) \in \mathscr{U}_{HL}$, $\mathcal{D}(\mathcal{HL}(L)) = \mathcal{HPN}$ is defined as following:

- $PN_{N0} = (PN_0, \mathcal{N}_0)$ such that: $PN_0 = \alpha(mLog_0)$ where α represents a process discovery algorithm; and $\forall t \in T_0$, $\mathcal{N}_0(t) = N$ if $\exists nec \in NE_H^C(mLog_0) : l(t) = nec$, $\mathcal{N}_0(t) = A$ otherwise; and
- $HPN(PN_{N0}) = \emptyset$ if $T_{n0} = \emptyset$; otherwise
- $HPN(PN_{N0}) = \{(t_i, PN_{Ni}, HPN(PN_{Ni}))|t_i \in T_{n0}\}$ where $PN_{Ni} = (PN_i, \mathcal{N}_i)$ such that $PN_i = \alpha(ILog_{nec})$ where $nec \in NE_H^C(mLog_0)$ and $nec = l(t_i)$; and $\forall t \in T_i$, $\mathcal{N}_i(t) = N$ if $\exists nec' \in NE_H^C(ILog_{nec}) : l(t) = nec', \mathcal{N}_i(t) = A$ otherwise.

An example of a hierarchical Petri net is shown in Fig. 5. It can be obtained from the example execution trace σ_{C1}. The top-level PN_N contains one normal transition (*class2.init()*) and two nested transitions each referring to a Petri net with nested transitions. More specifically, nested transition *class1.init()* refers to PN_{N1} and *class1.perform()* calls PN_{N2}. Because both PN_{N1} and PN_{N2} do not contain any nested transitions, the recursive definition stops at this level.

V. Implementation in ProM

The open-source (Pro)cess (M)ining framework *ProM 6* [18] has been developed as a completely plugable environment for process mining and related topics. It can be extended by adding plug-ins, and currently, more than 1500 plug-ins are included. The framework can be downloaded freely [2].

The proposed software component behavior discovery approaches have been implemented as two consecutively executed plug-ins in our *ProM 6* package [3]. The first one, called *Identifying Software Event Log of Components*, is used to generate a set of software event logs for different components by taking (1) the original software execution data, and (2) a configuration file describing which classes belong to which components. The second plugin, *Software Component Behavior Discovery*, takes the software event log of each component as input, and returns its behavioral model.

VI. Experimental Analysis

In this section, we use an online bookstore software as a case to show that the proposed approach exploits both component information and hierarchical structure and as a result produces more understandable behavioral models. This case contains three components, i.e., *Starter*, *SearchOffer* and *OrderAndDelivery*. The *Starter* component contains *BookstoreStarter* class, the *SearchOffer* component consists *Catalog*, *BookSeller* and *Bookstore* classes, and the *OrderAndDelivery* component contains *Order*class and *Delivery* classes. More specifically, the software starts with the instantiation of the *Starter* component, the *SearchOffer* component is used to get an offer for each book and the *OrderAndDelivery* component is used to generate book orders and perform delivery for all selected books. We first instrument the source code using *Kieker* framework [19]. Therefore, the method invocations are stored as software execution data. Each method call has all attributes defined in Definition 3. In this experiment, we created a software execution data by collecting 20 traces which cover all possible execution scenarios.

Without considering component information and hierarchical structure of the software, we can directly transform the original software execution data (i.e., without using the approach described). This yields a software event log with 20 cases, denoted as *OnlineBookstore.xes*, where each case refers to an execution trace. Taking this event log as input, we run the *Inductive Miner* with default settings and discover a behavioral model as shown in Fig. 6. It is a flat Petri net model where the labeled transitions represent methods and the places represent method invocation relations. The model is a bit spaghetti-like and contains misleading behavior which hinders the understanding of how software really behaves. For example, the model has some self-loops which do not exist in our software implementation. Using our approach, we split the software execution data to different components, and a set of software event logs each refers to one component are obtained.

[2]http://www.processmining.org

[3]https://svn.win.tue.nl/repos/prom/Packages/CongLiu/

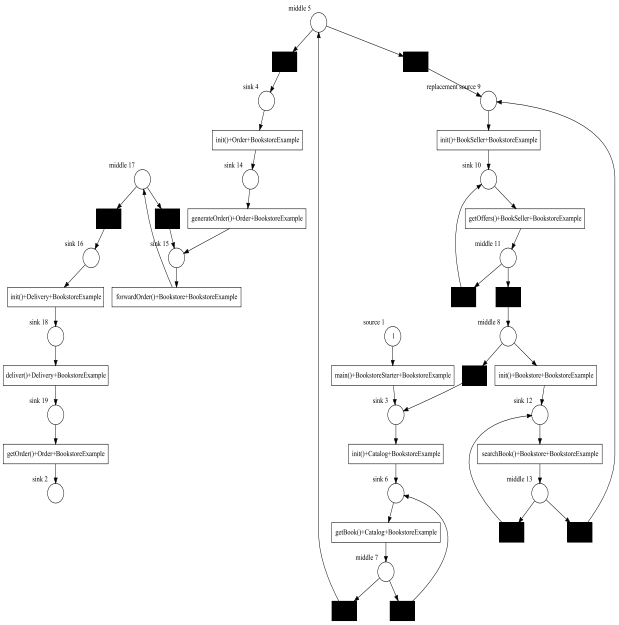


Fig. 6. Behavioral Model of Online BookStore Software without Hierarchy

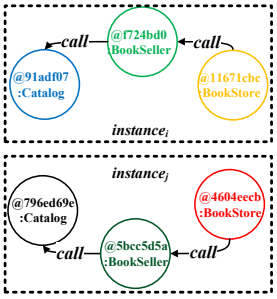


Fig. 7. Class Object Interaction Graph of one Execution Trace

In addition, one software execution may trigger multiple executions of a component, i.e., multiple component instances. For example, the instance number of component *SearchOffer* is the same with the number of searched books as this component is iteratively instantiated for each book. Fig. 7 gives an example of class object interaction graph of one *SearchOffer* component execution trace. It contains two weakly connected sub-graphs, each refers to one instance of *SearchOffer* component. Next, we refactor each component log by identifying component instances as new case notion using the approach in *Section* IV-B.

According to Table II, the refactored logs of different components are *Starter.xes* with 20 cases, *SearchOffer.xes* with 54 cases, and *OrderAndDelivery.xes* with 20 cases. The reason why *SearchOffer.xes* has more than 20 cases is that each execution will invoke an arbitrary number (exactly the number of searched books) of instances. Taking these refactored logs as inputs, we run the *Inductive Miner* again for different components and obtain one behavioral model per component as shown in Fig. 8. Fig. 8 shows three flat Petri nets each describing the behavior of one component. The behavioral model of each component is much simpler compared to that in Fig. 6. However, given the observation that software usually has a hierarchical structure implemented as multi-level nested method calls, the discovered flat models still cannot give an accurate description of its actual behavior.

Next, we take the notion of hierarchy into account and perform hierarchial behavior discovery for each component

TABLE II
SOFTWARE EVENT LOG SIZE COMPARISON

Log Name	*Number of Cases*
OnlineBookStore.exs	*20*
Starter.xes	*20*
SearchOffer.xes	*54*
OrderAndDelivery.xes	*20*

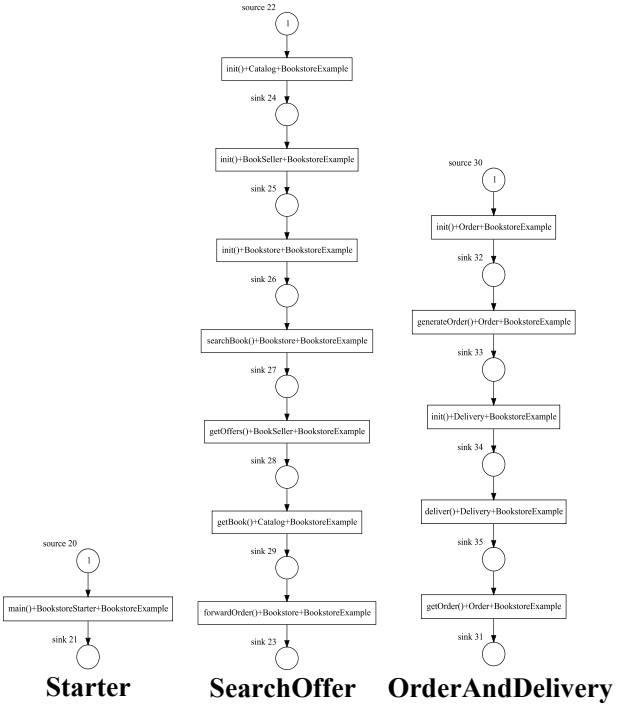


Fig. 8. Behavioral Models of Different Components

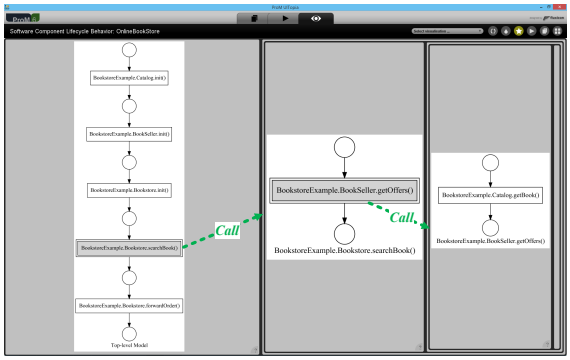


Fig. 9. Behavioral Model of the SearchOffer Component

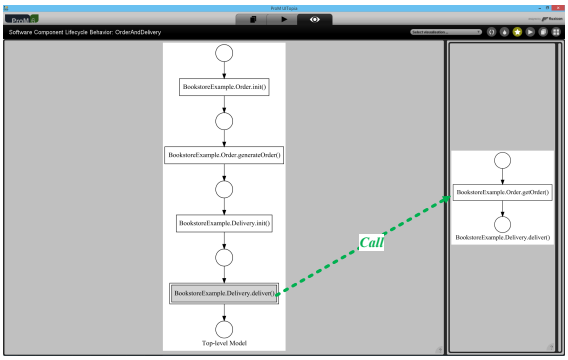


Fig. 10. Behavioral Model of the OrderAndDelivery Component

using the approach in *Section* IV-C. The discovered hierarchical behavioral models of *SearchOffer* and *OrderAndDelivery* are shown in Figs. 9-10 where: (1) a single-line rectangle represents an atomic method call; and (2) a double-line rectangle stands for a nested method call which refers to another sub-net.

By comparing the flat model in Fig. 6 with the hierarchical models of each component, we argue that the use of component information and hierarchy helps to give a better understanding of how software behaves from the perspective of individual component.

VII. Conclusion

By exploiting tremendous amounts of software execution data, this paper proposes to utilize process mining techniques to discover behavioral models for each software component. To do this, we first identify component instances and construct software event logs for each component from the raw software execution data. Then, based on the software event log, we construct its corresponding hierarchical software event log using calling relations among methods. Next, the software behavioral model, represented as a hierarchical Petri net with nested transitions, is discovered from a hierarchical software event log by recursively applying existing process discovery techniques. Our proposed approaches are demonstrated through an online bookstore software case and implemented in the open source process mining toolkit ProM.

This work serves as a starting point for several research directions. Given a software with several components, these components usually interact with each other by inter-component method invocations. The representation and discovery of inter-component interactions are our on-going work. A component is a set of classes collaborating to perform a particular function. This paper assumes that the mappings between classes and components are known beforehand. In real-life cases, for instance some legacy software systems, the development documents are incomplete or even unavailable. Automatic identification of components (groups of classes) from software execution data will be investigated in the future.

References

[1] W. v. d. Aalst, "Big software on the run: in vivo software analytics based on process mining (keynote)," in *Proceedings of the 2015 International Conference on Software and System Process*. ACM, 2015, pp. 1–5.

[2] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE transactions on Computers*, no. 6, pp. 592–597, 1972.

[3] 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*, vol. 32, no. 9, pp. 642–663, 2006.

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

[5] S. D. Kim and S. H. Chang, "A systematic method to identify software components," in *Software Engineering Conference, 2004. 11th Asia-Pacific*. IEEE, 2004, pp. 538–545.

[6] M. Leemans and W. M. 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*. IEEE, 2015, pp. 44–53.

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

[8] J. Li, R. J. C. Bose, and W. M. van der Aalst, "Mining context-dependent and interactive business process maps using execution patterns," in *Business Process Management Workshops*. Springer, 2010, pp. 109–121.

[9] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 345–354.

[10] X. Lu, M. Nagelkerke, D. v. d. Wiel, and D. Fahland, "Discovering interacting artifacts from erp systems," *Services Computing, IEEE Transactions on*, vol. 8, no. 6, pp. 861–873, 2015.

[11] M. McGavin, T. Wright, and S. Marshall, "Visualisations of execution traces (vet): an interactive plugin-based visualisation tool," in *Proceedings of the 7th Australasian User interface conference-Volume 50*. Australian Computer Society, Inc., 2006, pp. 153–160.

[12] H. Nagamochi and T. Ibaraki, *Algorithmic aspects of graph connectivity*. Cambridge University Press New York, 2008, vol. 123.

[13] W. Reisig, *Petri nets: an introduction*. Springer Science & Business Media, 2012, vol. 4.

[14] V. Rubin, C. W. Günther, W. M. Van Der Aalst, E. Kindler, B. F. Van Dongen, and W. Schäfer, "Process mining framework for software processes," in *Software Process Dynamics and Agility*. Springer, 2007, pp. 169–181.

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

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

[17] W. M. van der Aalst, A. Kalenkova, V. Rubin, and E. Verbeek, "Process discovery using localized events," in *International Conference on Applications and Theory of Petri Nets and Concurrency*. Springer, 2015, pp. 287–308.

[18] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst, "The prom framework: A new era in process mining tool support," in *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets*, ser. ICATPN'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 444–454.

[19] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the kieker framework," 2009.

[20] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 248–257.