Hindawi
Scientific Programming
Volume 2021, Article ID 4788357, 22 pages
https://doi.org/10.1155/2021/4788357

Research Article

A Discovery Method for Hierarchical Software Execution Behavior Models Based on Components

Yahui Tang[1], **Tong Li**[2], **Rui Zhu**[3], **Fei Du**[1], **Jishu Wang**[3] and **Zifei Ma**[4]

[1]*School of Information, Yunnan University, Kunming 650500, China*
[2]*School of Big Data, Yunnan Agricultural University, Kunming 650201, China*
[3]*School of Software, Yunnan University, Kunming 650091, China*
[4]*School of Water Conservancy, Yunnan Agriculture University, Kunming 650201, China*

Correspondence should be addressed to Tong Li; tli@ynu.edu.cn and Rui Zhu; rzhu@ynu.edu.cn

Received 29 April 2021; Revised 19 July 2021; Accepted 13 August 2021; Published 26 August 2021

Academic Editor: Francisco Ortin

Software is rapidly evolving and operates in a changing environment; therefore, in addition to software design and testing, it is essential to observe and understand the software execution behavior by modeling data recorded during the execution of the software to improve its reliability. The nested call relationship between methods during the execution of software is common, but most process-mining methods are unable to discover them, only generating flat models with low fitness. Meanwhile, it is easy to generate "spaghetti-like" models with low comprehensibility when dealing with complex software execution data. This paper proposes a component-based hierarchical software behavior model discovery method that can discover hierarchical nested call structures during software runtime, improving the fitness of the model; meanwhile, the proposed method partitions the discovery model into several parts by component information to improve the comprehensibility of the model, which can also reflect the interaction behavior within and between components. The proposed approach was implemented in a process mining toolkit. Using real-life software event logs and public datasets, we demonstrated that compared with other advanced process mining techniques, our approach can visualize actual software execution behavior in a more accurate and easy-to-understand way while balancing time performance.

1. Introduction

Software runs and evolves in a constantly changing environment; some exceptions and crashes may occur during the execution of software. All the potential problems cannot be anticipated during software design. Therefore, in addition to software design and testing, observing and understanding the behavior of the software operating environment is essential [1]. Large volumes of data are recorded during the execution of an application, which provides valuable information on the software operation and interaction. The collection and analysis of these data can help us understand and improve software execution behavior.

Process mining can extract process information from event logs generated by information systems; based on this information, a concise, reasonable, and high-quality process model can be discovered to support the analysis and improvement of processes, thus providing new means for process discovery, supervision, and improvement in related fields [2]. With the abundance of process mining techniques and the improvements of software execution data availability, there has been widespread use of process mining technology for the analysis of software execution data. This interdisciplinary field of research is called software process mining [3], which analyzes software execution data from the perspective of the process. The application of process mining technology can model software execution data, capture software running behavior, and provide useful insights on software behavior, ultimately improving the usability and redesign of the software.

Software execution data consist of several events, each of which refers to a method call during software execution. During the execution of software, hierarchical nested calls are

common—that is, a method call invokes other methods while it is running [4, 5]. Most process mining methods can only generate flat (single-level) models in which all events are at the same level, which reduces the fitness of the model. A hierarchical model can better reflect software execution behavior, and the discovery of nest structures can avoid some loops or parallel structures, thus improving the simplicity of the model. The structure of the event log determines the structure of the process model; therefore, constructing a hierarchical event log is essential. Meanwhile, it is easy to generate a "spaghetti-like" model when dealing with complex data; such a model does not have a clear structure and exhibits low comprehensibility. Since software is typically composed of multiple components, partitioning the mined model by component information can make the model easy to understand. This paper presents a method that can discover process models with both component and hierarchical information to better reflect the execution behavior of the software. Meanwhile, the existing model metrics are only applicable to flat process models, and we extend them to hierarchical structures.

The main contributions of this paper are as follows:

(1) We construct the hierarchical event log to reduce the complexity of input and then propose a nested structure discovery algorithm based on the inductive miner (IM) algorithm to discover the nested call structure.

(2) The generated hierarchical model is partitioned by component information to improve its comprehensibility, enabling the interaction within components and between them to be discovered. The hierarchical process tree with components is proposed as the representation of the final process model.

(3) We extend two existing metrics to quantify the quality of hierarchical models. Experiments show that the proposed method can discover models with high fitness and simplicity in a more comprehensible way while balancing time performance.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 introduces the proposed method in detail. Section 4 presents the tool implementation and the experimental evaluation. The results and discussion are presented in Section 5. Finally, conclusions with a summary are drawn in Section 6.

2. Related Work

This study aims to discover a software execution behavior model from software execution data, positioned at the intersection of software dynamic analysis [6] and process mining—that is, using the process mining method to analyze software execution behavior. In this section, we review existing work on software dynamic analysis, process mining, and software process mining.

2.1. Software Dynamic Analysis. The research object of software dynamic analysis is a software system running on an actual or virtual processor, and software execution behavior is modeled from the execution data. Most dynamic analysis methods [7–9] cannot discover concurrency, thus resulting in complex models with an explosion of states. Meanwhile, some dynamic analysis methods unable to aggregate information from multiple executions, which ultimately leads to the discovery of several behavior models.

2.2. Process Mining. Process mining is a technique used to discover, monitor, and improve real business processes by extracting knowledge from event logs [2]. Process mining methods have been effectively applied in many fields, including information systems, banking and insurance, government agencies, education, and health care [10, 11]. Discovery algorithms are designed to generate process models from event logs. Compared to software dynamic analysis, different quality dimensions including fitness [12] and simplicity [13] are used to measure the quality of process models. However, discovery algorithms based on Petri net and its extensions, e.g., [4, 14–16], are unable to guarantee soundness. The IM algorithm is a state-of-the-art process mining algorithm that can mine block-structured process models with a fitness of more than 80% in polynomial time [12]. Block-structured process models ensure soundness and can be abstracted into process trees with high expressiveness and readability [17]. However, all these algorithms including the IM algorithm are unable to discover hierarchical structure.

2.3. Software Process Mining. Software process mining is the application and extension of process mining technology in the field of software engineering. It includes three aspects: (1) the mining software development process [18]; (2) the mining software-user interaction [19]; and (3) the mining software execution behavior [20]. This study focuses on the third aspect. Software execution behavior discovery can extract knowledge from software execution data and provide useful insights on software behavior during execution.

The process mining method is generic and can be applied directly to the software execution event log. The difference is that software execution data may have specific properties, e.g., hierarchical. References [15, 21] are the early application of process mining techniques in the software field, which focuses more on discovering formal and accurate models. However, these two methods could not handle hierarchical behavior. Compared with the traditional process mining and the early software process mining technology, the recently proposed software process mining methods pay more attention to the discovery of hierarchical structure. Leemans et al. [5] extended the flat event log to the hierarchical event log and proposed a hierarchical process model discovery method that can guarantee the soundness of the model. However, this method could not discover component information. Liu et al. [4] divided the software execution log into several independent sublogs based on the component information; sublogs were then mined respectively to generate process models. However, the method generated independent component models for each sublog, which was

unable to discover the interaction between different components. In [22, 23], Liu et al. formally define the software event log and its transformation from original software execution data; then, an automatic discovery method for hierarchical process model from the software event log is proposed. This method identifies the nested structures by calculating the frequency ratio of basic ordering relations between events and generates a single hierarchical process model that can reflect the interaction within and between different components. However, the discovery of nested structures greatly depends on the setting of the threshold, and the process model is represented based on the Petri net, which cannot guarantee soundness.

According to the issues based on existing technology, this study formally defines the software event log and add layer attribute and nested attributes to the event; the construction of the hierarchical structures is based on these two attributes. Then, we extend the IM algorithm to represent the hierarchical structure while ensuring the soundness of the model and improve the comprehensibility of the model with component information. The generated model can provide useful insights on software execution behavior.

2.4. Criteria for Comparison. For the comparison of different technologies, we define several features. Firstly, (a) an accurate and suitable model should have formal execution semantics and (b) the model quality should be measured. Including metrics like fitness and simplicity, (c) soundness [12] is defined as a model without deadlock or other anomalies. A sound model helps in the comprehension of a model, and an unsound model is unable to be measured by fitness. (d) The analysis should be statistically significant, so the technology should be able to aggregate information across multiple executions. Secondly, the model should be expressive and can capture the behavioral structure that may occur in the execution of software. Specifically, (a) the model should have the ability to express the hierarchical structure to support the nested calling relationships that are common in software runtime, (b) component behavior should be considered to improve the comprehension of a model, and (c) compared with choice and loop, some methods cannot discover concurrency. Table 1 summarizes the comparison.

3. A Discovery Method for Hierarchical Software Execution Behavior Models Based on Components

3.1. Approach Overview. Figure 1 shows an overview of the proposed method, where the input is software execution data and the output is a hierarchical software behavior model. First, software execution data are converted to software execution event logs by mapping and preprocessing steps. The entire software system is divided into several components by grouping classes in specific ways, adding component information for each event accordingly. Second, we construct a hierarchical event log referring to the detection result of nested call relationship between methods; then, the hierarchical process tree with components is discovered by the proposed method, and finally, we extend several quality dimensions to quantify the quality of the discovered hierarchical models.

3.2. Conversion from Software Execution Data to Software Execution Event Log. To apply process mining, data recorded in the format of an event log are a prerequisite. Therefore, we need to convert the raw software execution data (see Definition 2) into the XES-based [31] software execution event log in Definition 4. The software execution event log consists of a group of cases, each referring to an independent execution of a software system. A case consists of a set of orderly events; each event corresponds to a method call. Both cases and events have specific attributes, which are described by the following definitions.

Let S be a set, $\varnothing$ be an empty set, and $|S|$ denote the number of elements in set S. The power set of S is $P(S) = \{S' | S' \subseteq S\}$.

Definition 1 (event [14]). U_M is the set of all possible method calls, U_N is the set of all possible called methods, U_C is the set of component names of methods (including method calls and methods), U_T is the set of timestamps, U_A is the set of activities (an event is related to the execution of an activity), and U_{AT} is the set of attribute names.

For any method call $m \in U_M$ and attribute AN $\in U_{AT}$, $\#_{AN}(m)$ is the value of attribute AN for method m. m has the following standard attributes:

#calleeM(m) $\in U_M$ is the method call identifier of m. For this paper, we use the class name as a prefix to uniquely identify the method name.

#calledN(m) $\in U_N$ is the identifier of the method called m.

#timeStamp $\in U_T$ is the timestamp of m.

#calleeC(m) $\in U_C$ is the component name to which m belongs.

#Nested(m) $\in$ [0, 1]: if m calls other methods, the value of this attribute is 1, and the opposite is 0.

Definition 2 (software execution data [4]). The software execution data (SD) are a finite collection of method calls, i.e., $SD \subseteq U_M$. $\sigma \subseteq U_M$ is a software execution trace: $\forall \sigma_i, \sigma_j \in SD$: $\sigma_i \cap \sigma_j = \varnothing \vee \sigma_i = \sigma_j$; i.e., software execution data consist of a group of execution traces, each describing an interaction between components.

Let $COM \subseteq P(UC)$ be the component set of a software system: $\forall C_i, C_j \in COM$, $C_i \cap C_j = \varnothing$ or $C_i = C_j$; i.e., there is no class sharing between different components and components of the same software cannot overlap.

Definition 3 (nested relation). In computer programming, a nested method is a method that is defined within another method, the enclosing method. In the software execution data, nested methods are recorded in the attribute called method.

Table 1: Comparison of related techniques and the expressiveness of the resulting models.

	Author	Algorithm/ technique	Formalism	Execution semantics	Model quality	Soundness	Aggregate runs	Hierarchy	Component behavior	Concurrency	Loop	Choice
Dynamic analysis	Walkinshaw [24]	Finite-state models	EFSM	✓	✓	n/a	✓	—	—	—	✓	✓
	Briand et al. [25]	Meta models	UML SD	✓[1]	—	n/a	—	—	—	—	✓	✓
	De Pauw et al. [26]	Execution patterns	Exec. pattern	—	—	n/a	—	✓	—	—	✓	—
	Beschastnikh et al. [27]	Synoptic	CFSM	✓	✓	n/a	✓	—	—	✓	✓	✓
	Heule and Verwer [28]	DFAsat	DFA	✓	✓	n/a	✓	—	—	—	✓	✓
Process mining	Van der Aalst et al. [14]	Alpha algorithm	Petri net	✓	—	—	✓	—	—	✓	✓	✓
	Leemans et al. [12]	Inductive miner	Process tree	✓	✓	✓	✓	—	—	✓	✓	✓
	Weijters and Verwer [29]	Heuristics miner	Heuristics net	✓	—	—	✓	—	—	✓	✓	✓
	Gunther and Van Der Aalst [30]	Fuzzy miner	Fuzzy model	—	—	—	✓	✓[2]	—	—	✓	✓
	van der Werf et al. [16]	ILP miner	Petri net	✓	✓	—	✓	—	—	✓	✓	✓
Software process mining	Leemans et al. [15]	Distributed systems	Petri net	✓	—	—	✓	—	—	✓	✓	✓
	Leemans et al. [5]	Recursion aware	Hierarchical process tree	✓	✓	✓	✓	✓[3]	—	✓	✓	✓
	Liu [4]	Component-based	Hierarchical Petri net	✓	✓	—	✓	✓[3]	✓[4]	✓	✓	✓
	Liu [22]	Component-based (single model)	Hierarchical Petri net	✓	✓	—	✓	✓[3]	✓[5]	✓	✓	✓
	This paper	Component-based process tree	Hierarchical process tree with components	✓	✓	✓	✓	✓[3]	✓[5]	✓	✓	✓

[1]Formal semantics are available for UML SD variants. [2]The hierarchy is based on anonymous clusters in the resulting model. [3]The hierarchy is based on the hierarchical information in the event log. [4]Generating independent component models for each sublog, being unable to discover the interaction between different components. [5]Discovering the internal execution and interaction behavior of components.

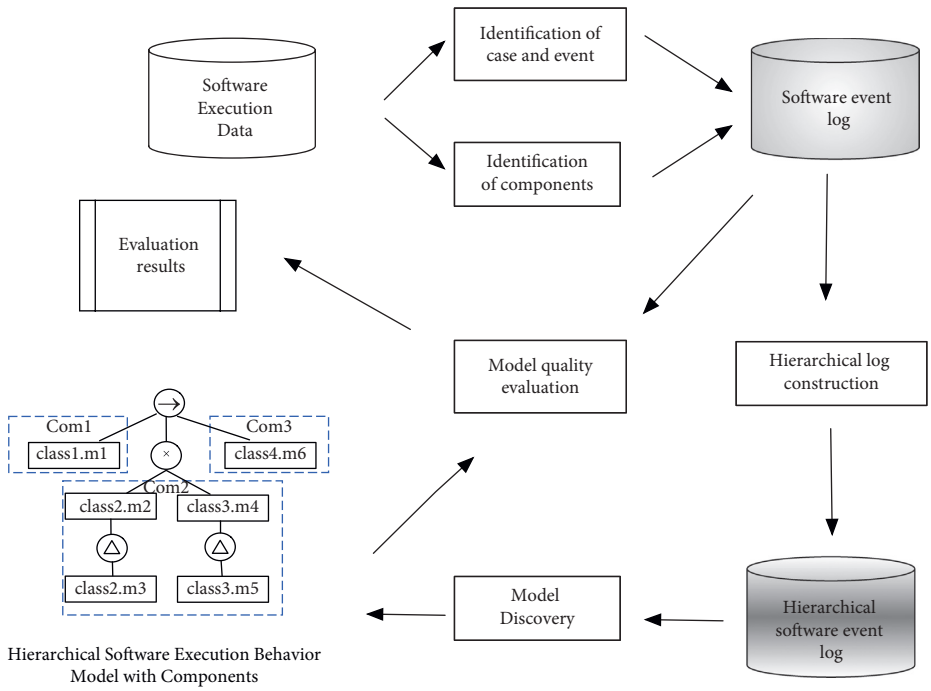


Figure 1: Approach overview of a discovery method for hierarchical software execution behavior models based on the component.

Definition 4 (case, trace, and software execution event log). C is the set of all possible case identifiers. A trace is a finite set of events $\sigma \in \varepsilon^*$, and each event occurs once; i.e., for $1 \le i \le j \le |\sigma|$, there is $\sigma_i \ne \sigma_j$. A software execution event log is a set of cases $L \subseteq C$, and each trace can appear at most once in the log—that is, for any $l_1, l_2 \in L$: $l_1 \ne l_2$: $\partial_{\text{set}}(l_1) \cap \partial_{\text{set}}(l_2) = \varnothing$.

Figure 2 shows the mapping from the software execution data to the event log. Figure 2 (right) shows the software execution data, consisting of a group of cases, each case represents an independent execution of the software. The software execution case is a collection of method calls, each of which has several attributes describing its execution information, including method call: the name of the method call; component: indicates which component the method call belongs to; called method: the name of the called method; timestamp: indicates when the method call occurs. Figure 2 (left) shows the corresponding software execution event log, where (1) an event log contains a collection of cases; (2) a case is a collection of events, and each event refers to a method call; (3) the event attributes and the method call attributes of software execution data have a one-to-one correspondence, whereas the nested attribute indicates whether one method calls another method, as defined in Definition 3. The attribute is obtained by detecting whether the called method identifier of the event in the software execution data is empty.

Table 2 shows the software execution data corresponding to the software system shown in Figure 3. Based on the mapping method shown in Figure 2, the software execution event log is shown in Table 3. Component information is obtained by grouping classes using the method presented in the next section.

3.3. Component Identification. A software system comprises a set of interacting components. A component is a kind of nontribal software entity that has relatively independent functions and can be deployed independently and replaced [22]. The term component is generic and can be composed of a set of classes. According to the execution data, we can obtain insight into how classes aggregate to form components [32, 33]. As shown in Figure 3, this software system consists of three components: com1, com2, and com3, and each component contains specific classes: com1 consists of class1, com2 consists of class2 and class3, and com3 consists of class4.

Component identification aims to group classes based on their components and then add component attributes for all events. We focus on the discovery process model from the event log with component information, not on the component identification method. The discovered model is divided into several regions based on the component information to improve its comprehensibility.

Community detection [34] can be used to identify the components. In [35], different community detection algorithms were evaluated using over 100,000 method calls recorded from the execution of four real software systems. The results showed that a multilayer local search algorithm of modular clustering proposed by [36] could more efficiently divide the entire software system into high-quality components. We used this method to identify components.

To better represent the execution behavior of software, we propose to represent the model as a hierarchical process tree with components (see Definition 7), by adding two elements to the traditional process tree: (1) components, indicating the block clusters representing different components and (2) hierarchy, that is, the nested call

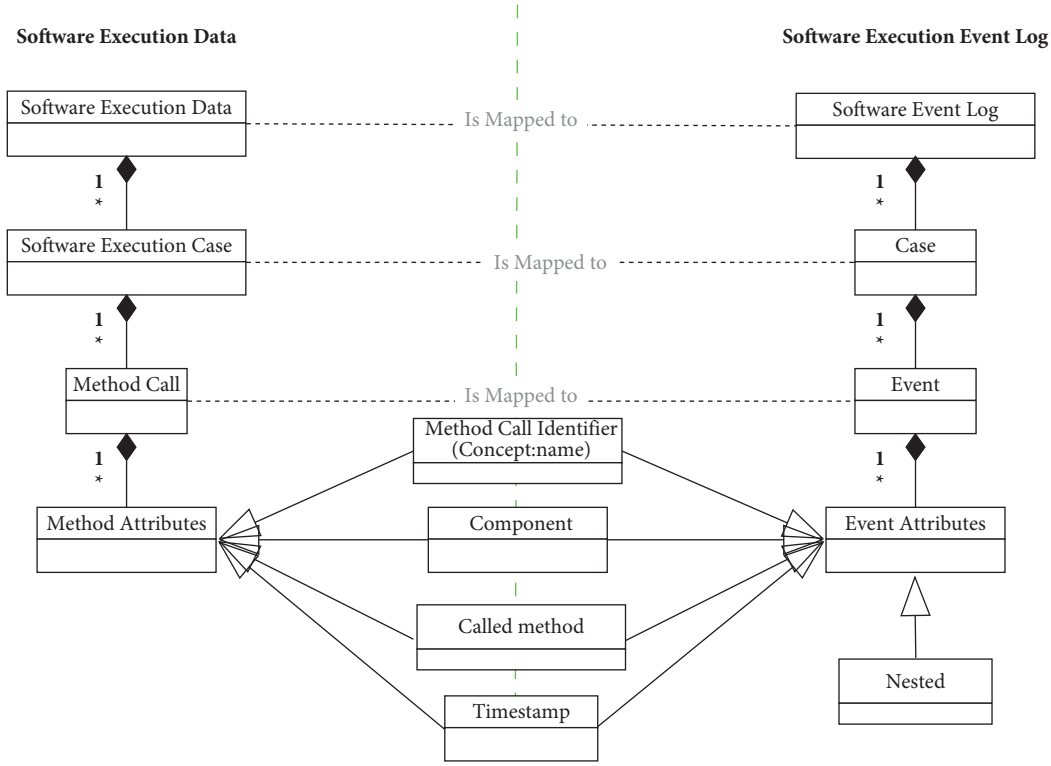


Figure 2: Mapping of software execution data to software execution event log.

Table 2: Software execution data corresponding to the software system example shown in Figure 3.

Execution identifier	Caller method identifier	Called method identifier	Component	Timestamp
1	class1.m1	NULL	com1	2021100076
1	class2.m2	class2.m3	com2	2021100077
1	class2.m3	NULL	com2	2021100078
1	class4.m6	NULL	com3	2021100079
2	class1.m1	NULL	com1	2021100081
2	class3.m4	class3.m5	com2	2021100082
2	class3.m5	NULL	com2	2021100083
2	class4.m6	NULL	com3	2021100084

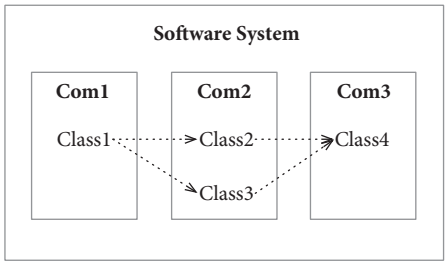


Figure 3: Architecture of a component-based software system.

relationship. Both component and hierarchical information can be obtained from the hierarchical event logs.

3.4. Construction of the Hierarchical Software Execution Event Log. The structure of the event log determines the structure of the mined model. To generate a hierarchical process model, it is necessary to construct a hierarchical software execution event log.

Definition 5 (hierarchical software execution event log, HLOG). Hierarchical software execution event log is a 3-tuple: HLOG= (mainLog, Mapping: event- > subLog, layer), satisfying mainLog which is the main log with a nested attribute that indicates whether it is a nested event. Mapping: event- > subLog is a mapping from nested method calls in mainLog to subLogs. Layer represents the number of layers of events.

This paper sets the event contained in mainLog to be layer 0, i.e., layer(mainLog) = 0, events called by events in mainLog are layer 1, events called by events in layer 1 are layer 2, and so on. Therefore, layer(subLog) > 0.

The key to obtaining a hierarchical event log is to detect the nested call relationship in the event log. Algorithm 1 can build a hierarchical event log that contains nested call relationships by nested detection. Algorithm 1 takes a software event log as input and outputs a hierarchical software event log, including (a) the mapping of nested method calls in mainLog to their corresponding sublog and (b) adding a

Table 3: Software execution event log corresponding to the software system example shown in Figure 3.

Case identifier	Concept:name	Nested	Called method	Component	Timestamp
1	class1.m1	0	NULL	com1	2021100076
1	class2.m2	1	class2.m3	com2	2021100077
1	class2.m3	0	NULL	com2	2021100078
1	class4.m6	0	NULL	com3	2021100079
2	class1.m1	0	NULL	com1	2021100081
2	class3.m4	1	class3.m5	com2	2021100082
2	class3.m5	0	NULL	com2	2021100083
2	class4.m6	0	NULL	com3	2021100084

Input: A software event log(*Log*);
Output: A hierarchical software event log (*HLog*).
1 *mainLog* ⟵ ∅; *Mapping* (*event, subLog*) ⟵ ∅.
2 *mainLog* ⟵ **getmainLog**(*log*);
3 *subLog* ⟵ **getSub**(*log*);
4 sublog.layer = *layer*+1;
5 *Mapping* (*event, subLog*) ⟵ **getMapping**(*Log*).
6 If *Mapping* (*event, subLog*) ≠ ∅ then
  For each *event* ∈ *Mapping* (*event, subLog*) do
    HlogConstruction (*subLog*).
    layer ++
  End do
 End if
7 *HLog* ⟵ *mainLog* ∪ *Mapping* (*event, subLog*) ∪*layer*.
8 return *HLog*.

Algorithm 1: HlogConstruction().

layer number to each event to show which layer the event belongs to.

Two external functions are included in Algorithm 1, among them, getmainLog() can return the mainLog of the software event log and label each event as nested or not nested; the external function getMapping() can return the mapping from nested events of mainLog to the subLog. To detect nested methods in all subLog, the function Hlog-Construction() recursively calls itself in step 6 by taking the newly generated subLog as input, and each recursive call adds 1 to the number of layers and adds the layer number to each event. And the recursion continues until Mapping: event- > subLog is empty.

Table 4 shows the hierarchical event log corresponding to the example software system shown in Figure 3.

3.5. Discovery of the Hierarchical Software Process Model. After constructing a hierarchical event log, a hierarchical process model can be generated. Compared with traditional languages such as Petri net, heuristic net, and fuzzy model [13], the process tree is the only one that can ensure soundness [13] because of the block structure. A sound model can facilitate an easy understanding of the behavior of software systems. Meanwhile, during the discovery of the hierarchical process model, we need to find the nested relationship between events of different layers and connect them to generate a complete model. The soundness of the process tree can avoid detecting and repairing the incomplete process model.

Definition 6 (process tree [12]). Process tree is defined recursively; let ⊗ be a finite set of operators, and tau $\notin A$ is the silent activity. Then, we have the following:

$a \in (A \bigcup \{\text{tau}\})$ is a process tree

Let $P_1, \ldots P_n$ $(n > 0)$ be the process trees and $\otimes \in \bigotimes$ be a process tree operator; then, $\otimes(P_1, \ldots, P_n)$ is a process tree

The following are the standard operators for the process tree:

× is the exclusive choice between one of the subtrees

⟶ is the sequential execution of all subtrees

|| indicates the parallel execution of all subtrees

$\circlearrowleft$ is a loop operator with loop body P_1 and optional loop paths $P_2, \ldots, P_n$ $(n \geq 2)$

The state-of-the-art IM algorithm can generate a process tree to ensure soundness, the model having advantages in terms of expressiveness and readability. While the process tree is flat, we expand the IM algorithm to handle the nested call relationship.

This paper defines a hierarchical process tree with components (HPTc) to represent hierarchical behavior. In addition to the structure of the process tree, nested operator

Table 4: Hierarchical software event log corresponding to the example software system shown in Figure 3.

Case identifier	Concept:name	Nested	Called method	Layer	Component	Timestamp
1	class1.m1	0	NULL	0	com1	2021100076
1	class2.m2	1	class2.m3	0	com2	2021100077
1	class2.m3	0	NULL	1	com2	2021100078
1	class4.m6	0	NULL	0	com3	2021100079
2	class1.m1	0	NULL	0	com1	2021100081
2	class3.m4	1	class3.m5	0	com2	2021100082
2	class3.m5	0	NULL	1	com2	2021100083
2	class4.m6	0	NULL	1	com3	2021100084

nodes connect methods to their nested calling methods, as well as mappings of activities to their components.

Definition 7 (hierarchical process tree with components, HPTc). A hierarchical process tree with components is 4-tuple: HPTc = ($PT_{mainLog}$, f, ComponentSet, $\mathscr{M}$) such that $PT_{mainLog}$ is the top-level process tree referring to mainLog. Let A be the set of all activities in the event log; i.e., $A = A_{normal} \cup A_{nested}$, A_{normal} is the collection of normal activities, A_{nested} is the collection of nested activities, and f is a mapping from $a \in A_{nested}$ to its nested calling activity $a \in A$.

In addition to the operators defined in Definition 5, the nested relationships between the two activities are connected by the nested operator△. ComponentSet is the set of components; $\mathscr{M}$ is a mapping from activity $a \in A$ to its belonging component $c \in$ ComponentSet.

Algorithm 2 takes HLog as input and returns an HPTc. It uses five external functions: IM() takes in mainLog to mine the top-level process tree using the IM algorithm; addOperator() adds nested operators for nested events; getNested() obtains the invoked events and connects them using nested operators; MapCom() maps all activities in the generated model to their corresponding components. Step 2 detects each event in the log and connects them to the events they invoke. The function NestedModelDiscover() recursively calls itself by taking the newly generated set of invoked events as input until the deepest number of layers is detected; all events can be connected to the events it contains. Step 4 maps each activity to its corresponding component and generates a hierarchical process tree with components.

Figure 4 is the model discovered by the IM algorithm corresponding to the example software system shown in Figure 3. Figure 5 shows the hierarchical process tree with components discovered by the proposed method using the same software system. As shown in the figures, the IM algorithm could not recognize the nesting relationship, and all events were grouped into the same layer, incorrectly identifying the nesting relationship between class3.m4 and class3.m5, class2.m2 and class2.m3 as sequential. The proposed method uses the IM algorithm to generate the top-level process model and then adds subactivities for each layer by nested operators based on nested relationships. Simultaneously, each activity in the process model can be partitioned into its corresponding component areas.

4. Tool and Experiments

4.1. Tool Implementation. Based on the proposed method, we have implemented two plug-ins in ProM. ProM [37] is an open-source process mining toolkit that can be extended by adding plug-ins. There are currently more than 1600 plug-ins available. The first plug-in, *Hlog construction*, can transform the software event log into a hierarchical software event log. The second one is *HPTc discovery*, which can generate a hierarchical process tree with components from hierarchical software event log. Figure 6 presents the interface of *HPTc discovery*, and the experimental evaluation is based on these two plug-ins.

4.2. Experimental Preparation. Experiments are divided into two parts. In Part 1, we compare the structure of the model generated by our method with other related technologies. In Part 2, we measure two aspects for several techniques: the model quality and the running time of the technique.

The experimental data of Part 1 are collected from three real software using the Kieker framework [38], including an online bookstore software system, a sport lobby website, and an academic community. The online bookstore software system provides a platform for customers to order books online. Sport lobby is a sports news website that provides various types of sports news. The academic community is an academic forum for researchers who provide academic news that users can subscribe to by registering with a public mail server. According to the component identification method, different components and the classes they contain are shown in Table 5. Each case of the event logs corresponded to an independent execution of the system and covered all possible software execution paths. All source code and event logs can be obtained from the following website: https://github.com/TangYahui9596/Experiment-Data.

The experimental data of Part 2 contain three additional public datasets: the JUnit 4.12 [39], Apache commons crypto 1.0.0 [40], and NASA CEV [41]. The size of event logs is usually quantified by four metrics: the number of events, number of activities (size of the alphabet), number of cases, and average case length [5]. The size of event logs is shown in Table 6.

Input: A hierarchical software event log (*HLog*).
Output: A hierarchical process tree with components (*HPTc*);
1 If *HPTc* = ∅ then
  HPTc ⟵ **IM**(mainLog);
  HLog ⟵ mainLog
 End if
2 For each *event* in *Hlog* do
  If *event.nested* = 1
   addOperator(*event*);
   nestedEvent ⟵ **getNested**(*event*);
   HPTc ⟵ **connect** (*event,* nestedEvent);
   nestedSet ⟵ nestedEvent
  End if
 End do
3 If *layer* < maxLayer
  HierarchicalModelDiscover(nestedSet)**;**
  layer **++;**
 else
4 For each *activity* in *HPTc* do
   HPTc ⟵ **MapCom**(*activity, component*);
  End for
 End if
5 return *HPTc.*

Algorithm 2: HierarchicalModelDiscover().

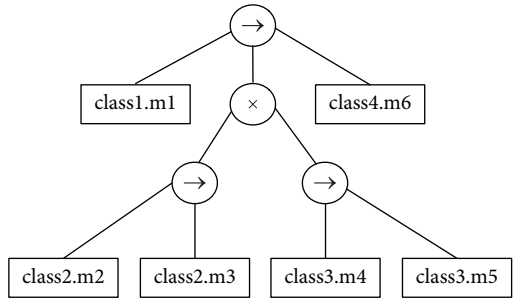


Figure 4: The mining result of the software system example in Figure 3 using the IM algorithm.

For these experiments, a laptop with a Quad-Core Intel Core i7 (1.7 GHz) CPU, running macOS Catalina 10.15.7 and Java SE1.7.0 67 (64 bit) with 16 GB of allocated RAM was used.

5. Results and Discussion

For Part 1, the contrasting approaches are the most classic and well-known process mining algorithm, alpha algorithm, state-of-the-art process mining algorithm IM, and the approach that can discover both component information and hierarchical structures [4], abbreviated as component based.

5.1. Correctness of the Mining Results. Figure 7 shows the UML sequence diagram of the online bookstore software system. The messages exchanged between classes show the nested method calling behavior and form of the hierarchical nature of the software. The sequence diagram provides a general understanding of the entire software execution behavior, and it can be used as a reference or ground truth to check the validity of the discovery results. More specifically, BookStore.searchBook() calls BookSeller.getOffer(), BookSeller.getOffer() calls Catalog.getBook(), and Delivery.delivery() calls Order.getOrder(). Figure 8(d) indicates that the model could correctly discover all these nested call relationships compared with the sequence diagram.

5.2. Model Structure Analysis. For Part 1, Figures 8–10 show the process model generated by different algorithms from three real-life software event logs. We can summarize the following from the figures.

The alpha algorithm and IM algorithm incorrectly identify events with nested calling relationships at the same level because they cannot discover the hierarchical structures, and some nested activities were identified as parallel, which makes the model more complex. Meanwhile, the generated process models contain several short loops that do not exist in the actual software execution. These loops can be explained since there are many cases in the event log, which makes it easy to identify the nested structure as a loop structure without hierarchical processing in the event of multiple cases. Finally, the model cannot find the relationship between activities and components.

The component-based method can discover hierarchical nested call structures and generate a hierarchical process model for each component in the form of a hierarchical Petri net. Concerning Figure 8(c), the method could correctly divide the software system into three components and generate a hierarchical process model for each of them. However, the three components were independent of each

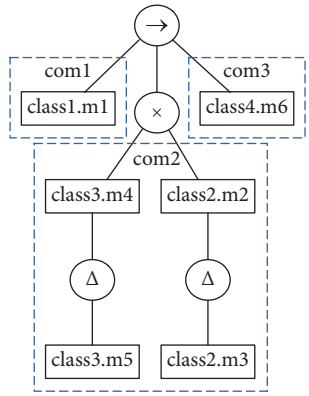


Figure 5: The mining result of the software system example in Figure 3 using the proposed method.

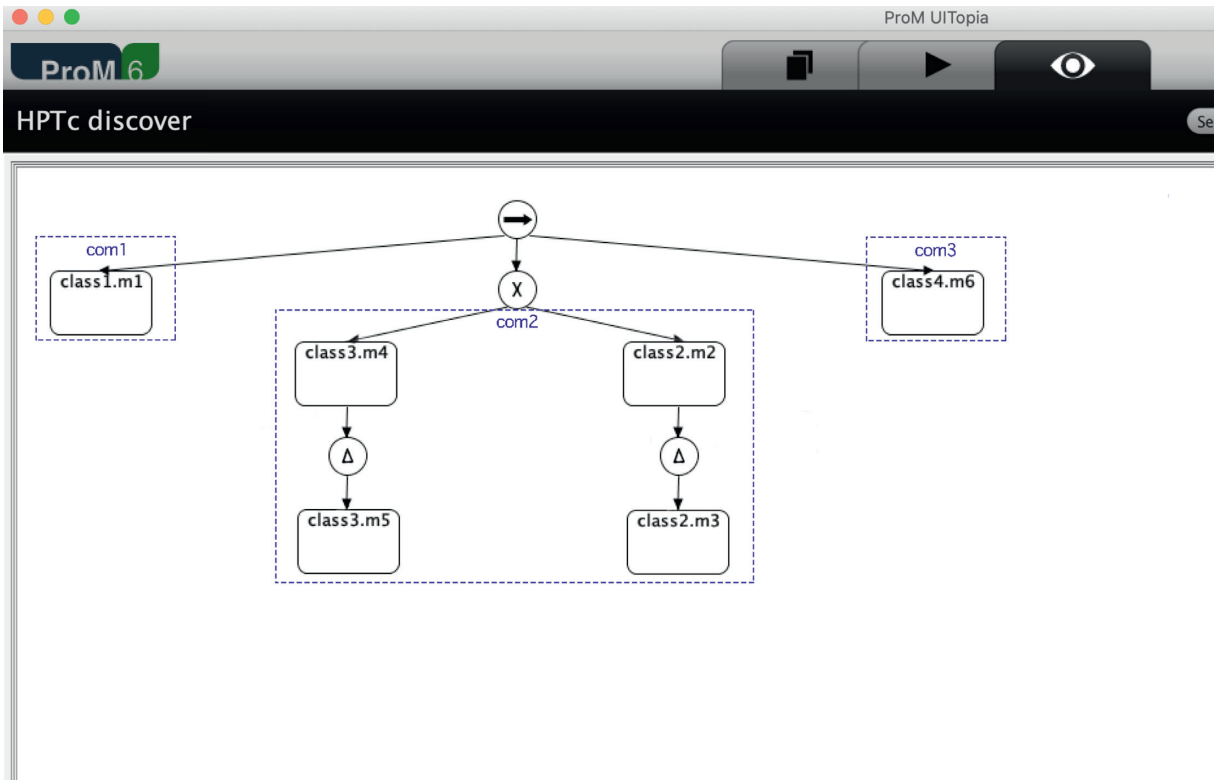


Figure 6: Interface of HPTc discovery plug-in.

Table 5: Classes included in components of the online bookstore software.

Software	Component	Included classes
Bookstore	Starter	BookstoreStarter
	SearchOffer	Catalog, BookSeller, bookstore
	OrderAndDelivery	Orderclass, delivery
Sport lobby	Starter	TestObserver
	User	SMSUser, Observer
	Commentary	Commentary
Academic community	Starter	TestMail
	MailingServer	MailingServer
	Member	Member

Table 6: The size of event logs used in the experimental evaluation.

Event log	No. of cases	No. of events	No. of acts	Avg. |case|
Bookstore	800	20 760	13	25.9
Sport lobby	200	6400	10	32
Academic community	618	15 450	7	25
JUnit 4.12	1	946	182	946
Apache	3	241973	74	80657.67
NASA CEV	2566	73638	47	28.70

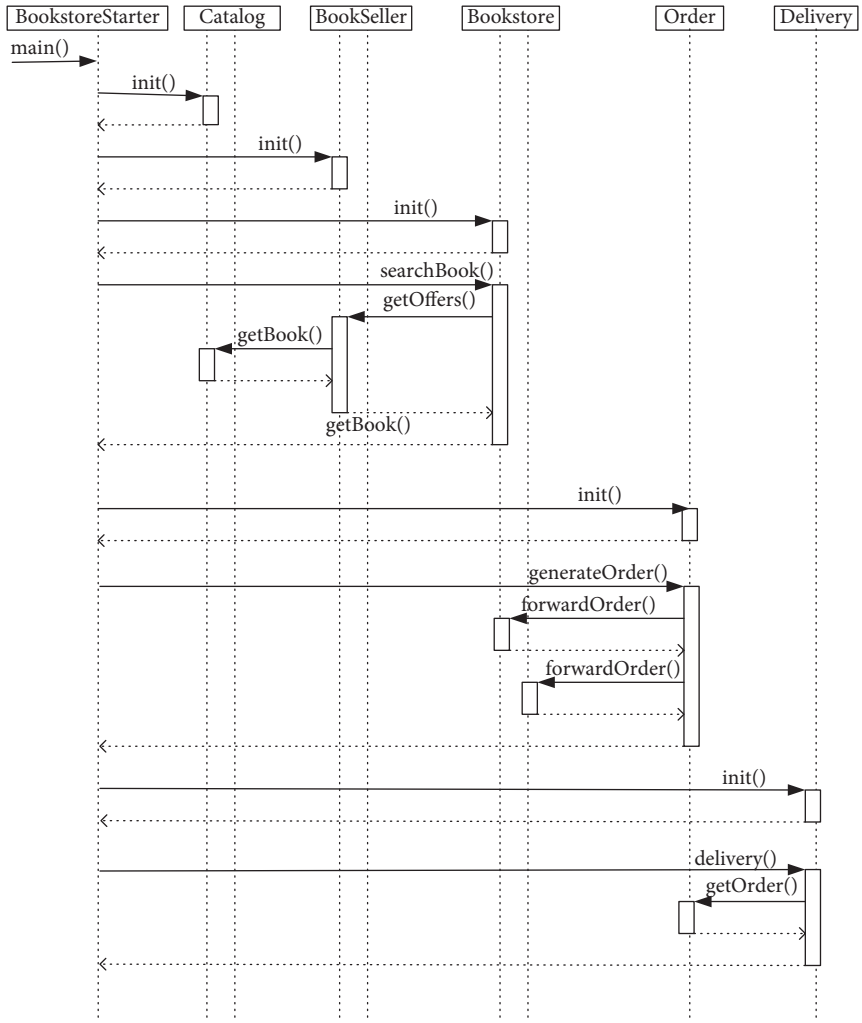
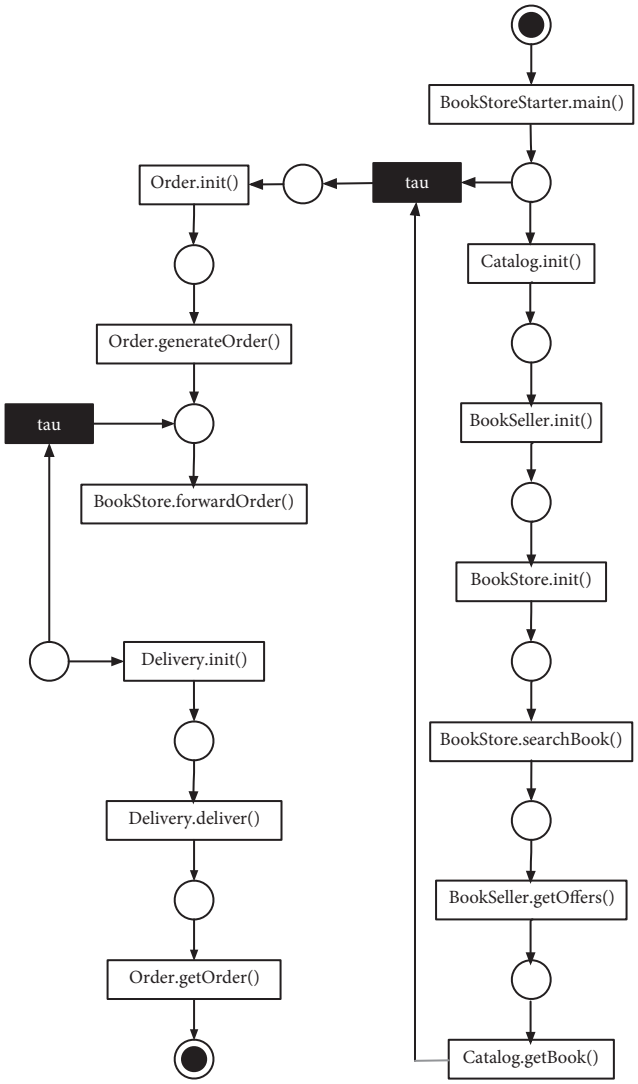


Figure 7: Sequence diagram of the bookstore software execution.

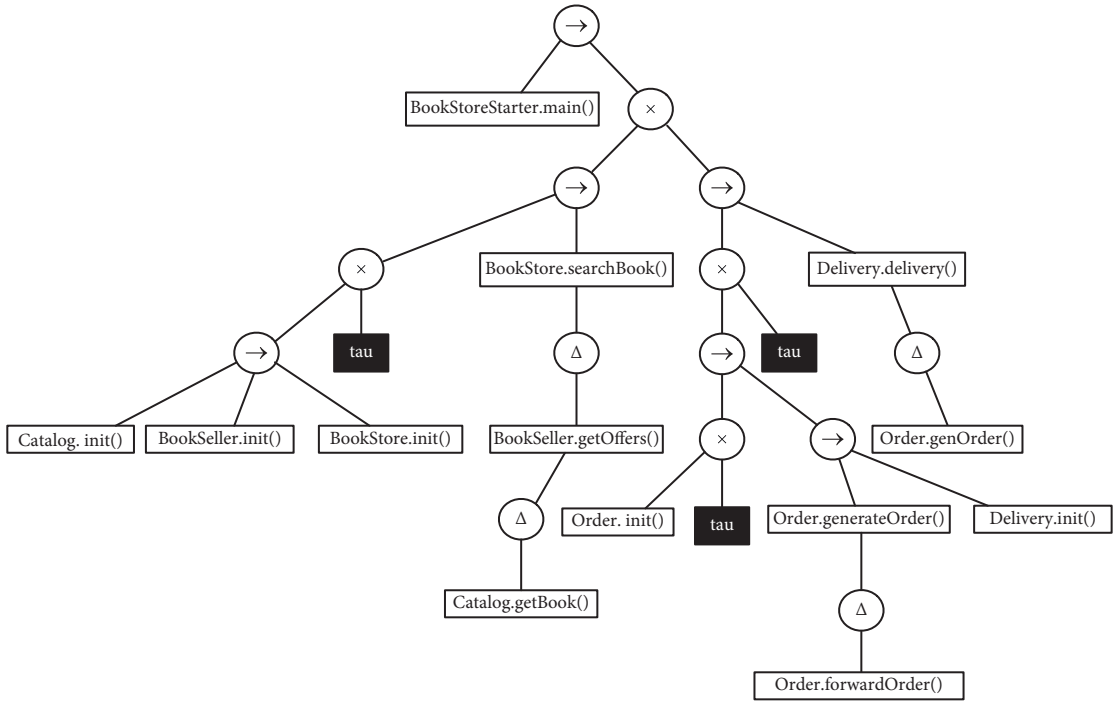
other, and the relationship between the components could not be discovered. Meanwhile, this method could only identify the internal hierarchical behavior of components; for example, in Figure 9(c), the Commentary.notifyObservers() belongs to the component Commentary nested call Observer.Update() method of the component User; therefore, the nested call structure could not be recognized, and some missing activities appear. Similarly, SMSUser.subScribe() and SMSUser.unsubScribe() belong to the component User; these methods nested called Commentary.subscribeObserver() and Commentary.unsubscribeObserver() separately, all of which belong to the component Commentary. In Figure 10(c), the MailingServer.notifyMember() method calls the methodsMember.init() method; they do not belong to the same component, and the method could not discover these nested relationships and result in missing activities in the model.

Figures 8(d), 9(d), and 10(d) show the process models discovered by the proposed method; they were divided into several components to improve comprehensibility. Among them, different blue block diagrams represent different components, indicating the name of each component. The divided model could reflect the internal behavior of the components and the interaction between

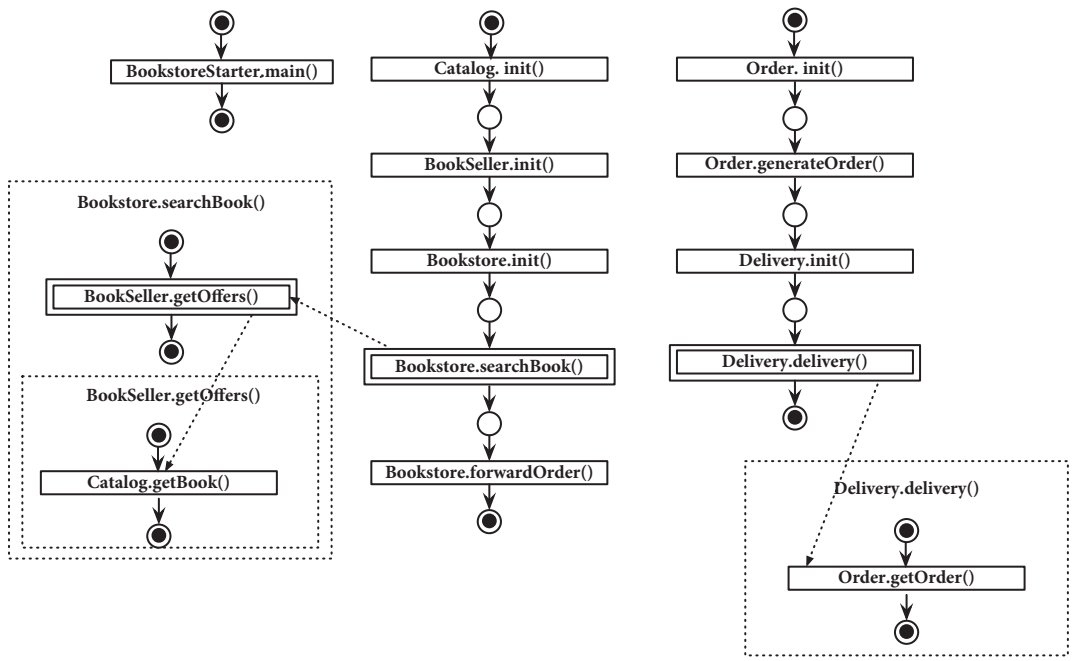


(a)

Figure 8: Continued.

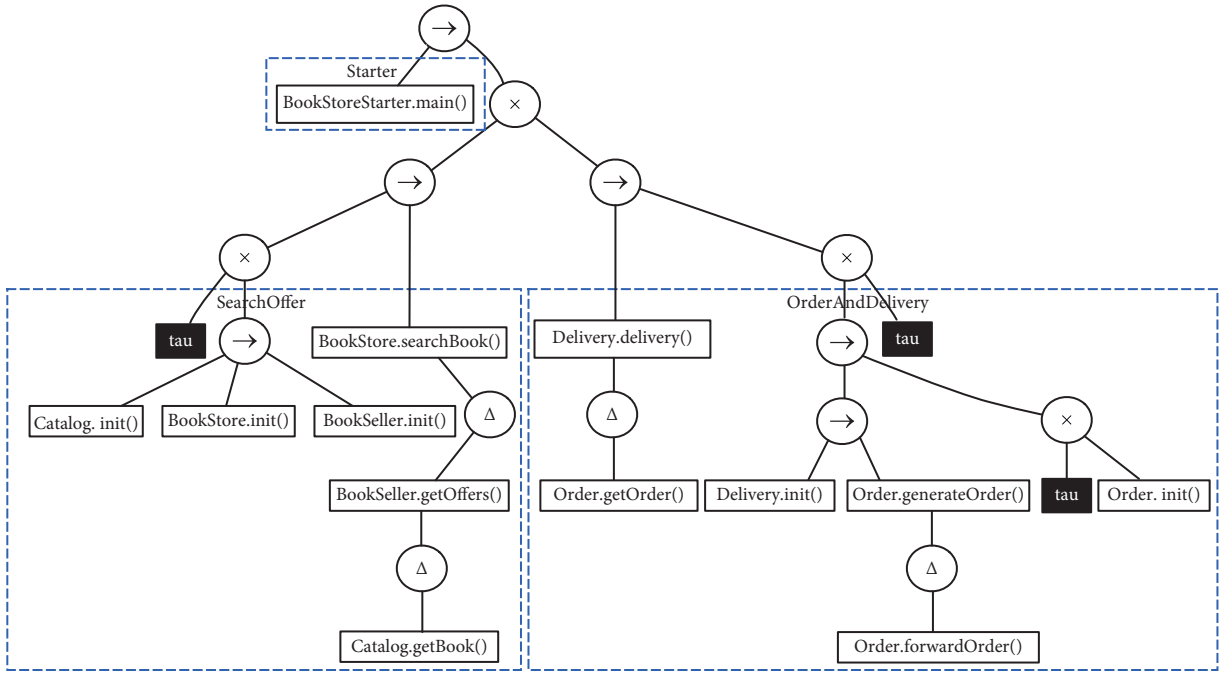


(b)



(c)

Figure 8: Continued.



(d)

Figure 8: Models discovered by different algorithms on bookstore data. (a) The alpha algorithm. (b) The IM algorithm. (c) The component-based method. (d) The proposed method.

them. Meanwhile, Figures 9(d) and 10(d) indicate that regardless of whether methods with nested structures belonged to the same component, the proposed method could recognize them and avoid the loss of activities in the model.

For Part 2, in addition to the algorithms mentioned in Part1, we add heuristic miner, ILP, component-based (single model), and fuzzy miner to compare the model quality and the running time of the technique.

5.3. Model Quality Analysis. Different quality dimensions have been proposed to measure the quality of the process. The most important of them are fitness and simplicity, the best model can explain all the behaviors in the event log (fitness), and it is also the simplest.

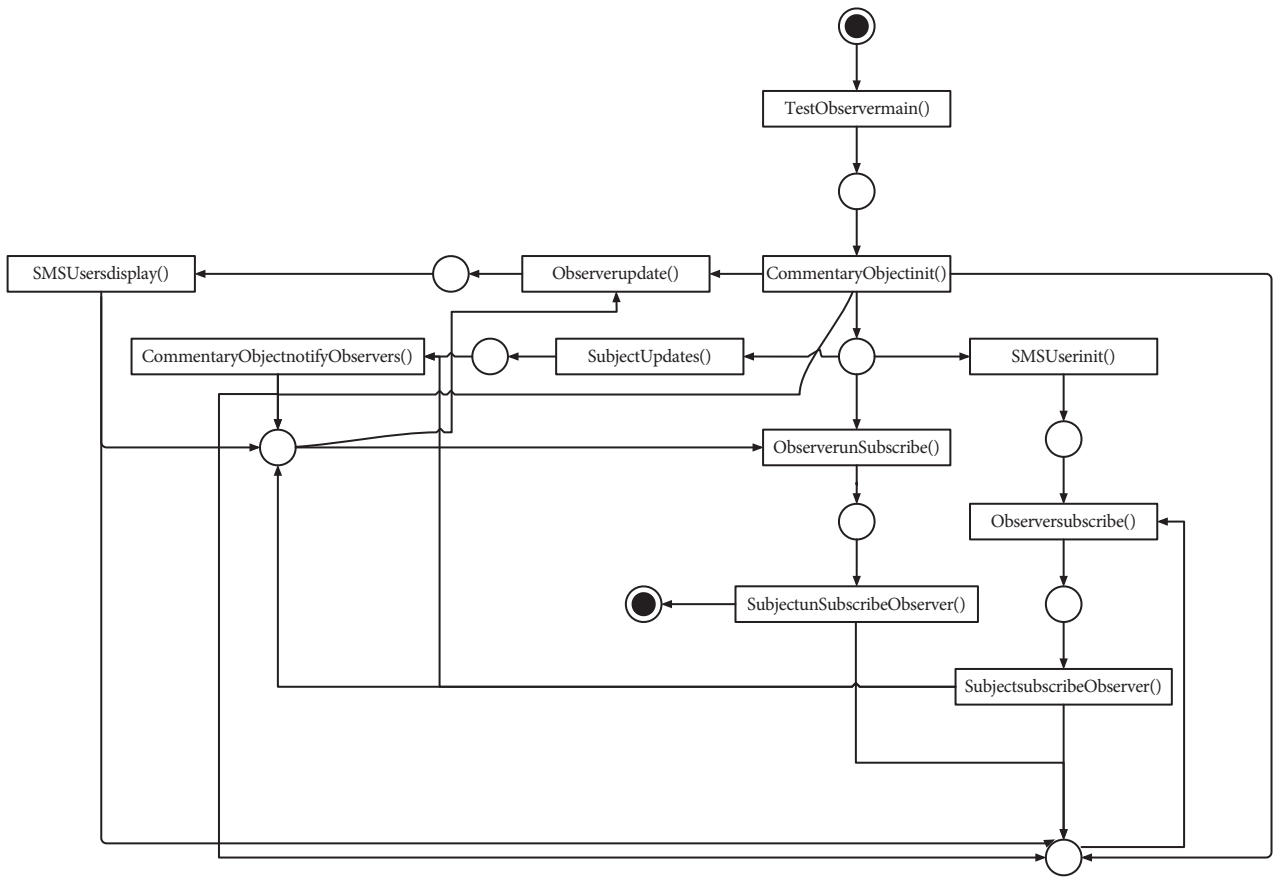
5.3.1. Fitness. Fitness reflects the extent to which the model can replay the event log. A perfect fitness value indicates that the model can replay all traces in the log. Current quantitative methods of fitness are only applicable to flat process models without nested structures. This study extends an alignment-based fitness computation defined in [13]; it aligns the event log with the process model to observe the inconsistency between them, which is calculated as Equation 1. *Min cost* is the minimal cost of aligning the event log to the model with no synchronous moves, and *Actual cost* is the actual cost for the aligning model and event log. The fitness value is in the range [0, 1].

$$\text{Fitness}_{\text{model}} = \frac{\text{Min cost}}{\text{Actual cost}}. \quad (1)$$

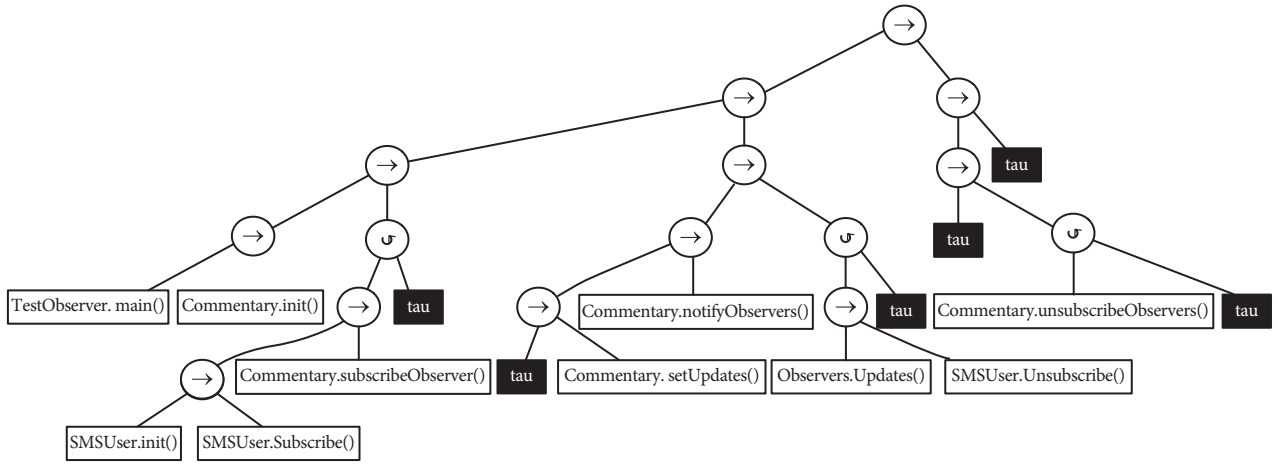
The calculation of fitness for a hierarchical process tree with components can be divided into two parts: the main model and the submodel. The main model is generated by mainLog that does not contain any nested structures, and the fitness of the main model can be calculated using (1). The submodel corresponds to the sublog that contains only nested structure, and we need to calculate its fitness separately.

$$\text{Fitness}_{\text{HPTc}} = \text{Fitness}_{\text{main model}} * \frac{\#\text{activities in main model}}{\#\text{total activities}} + \frac{\#\text{nested structure in model}}{\#\text{nested structure in log}} * \frac{\#\text{activities in sub} - \text{model}}{\#\text{total activities}}. \quad (2)$$

The fitness of the hierarchical process tree with components can be measured using (2), where $Fitness_{mainmodel}$ is the fitness value of the main model and (#nested structure in model/#nested structure in log) is the ratio of the nested structures correctly discovered in the model to all nested structures contained in the event log. *#activites in main model* is the number of activities in the main model, *#activites in submodel* is the number of



(a)



(b)

Figure 9: Continued.

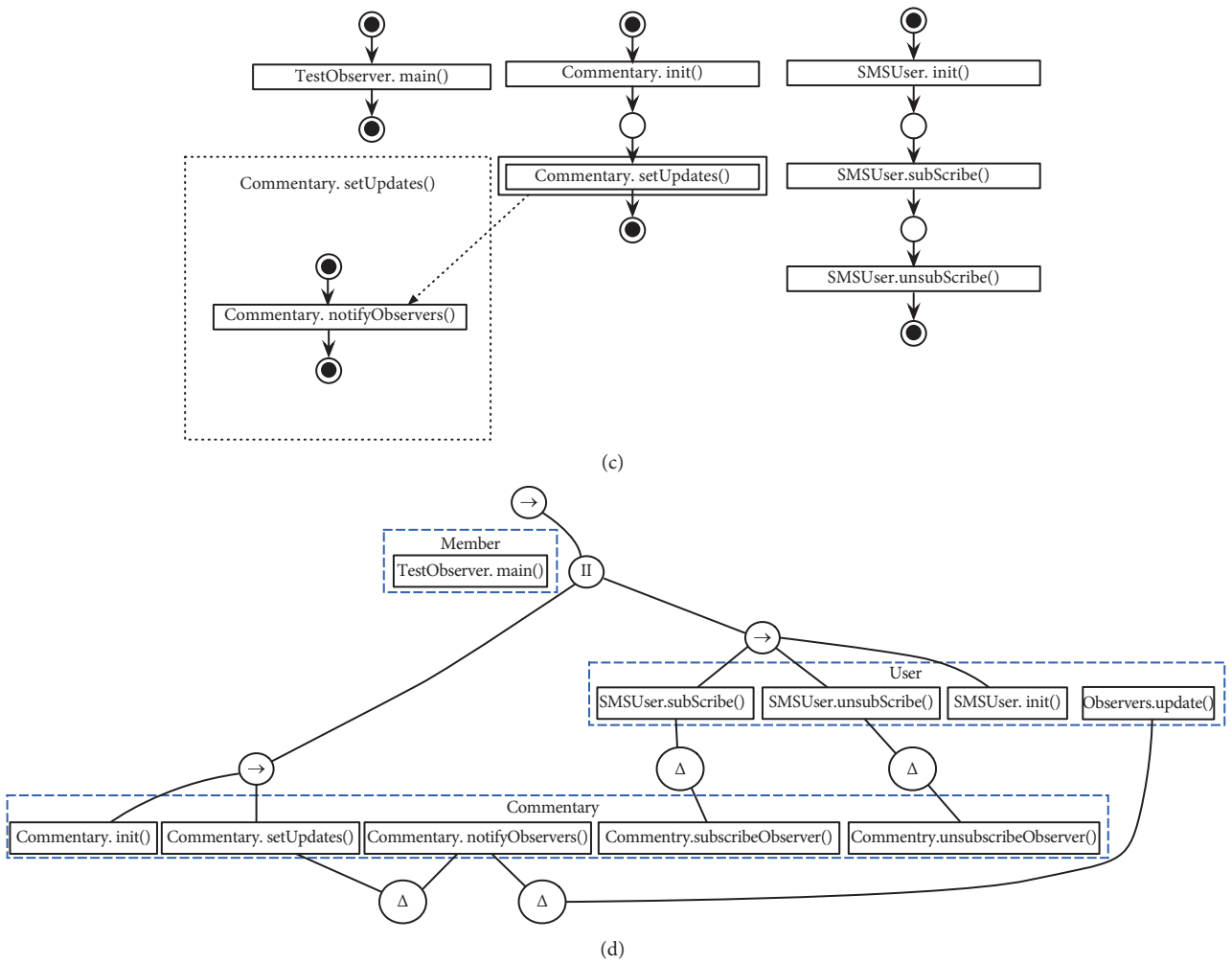


FIGURE 9: Models discovered by different algorithms on sport lobby data. (a) The alpha algorithm. (b) The IM algorithm. (c) The component-based method. (d) The proposed method.

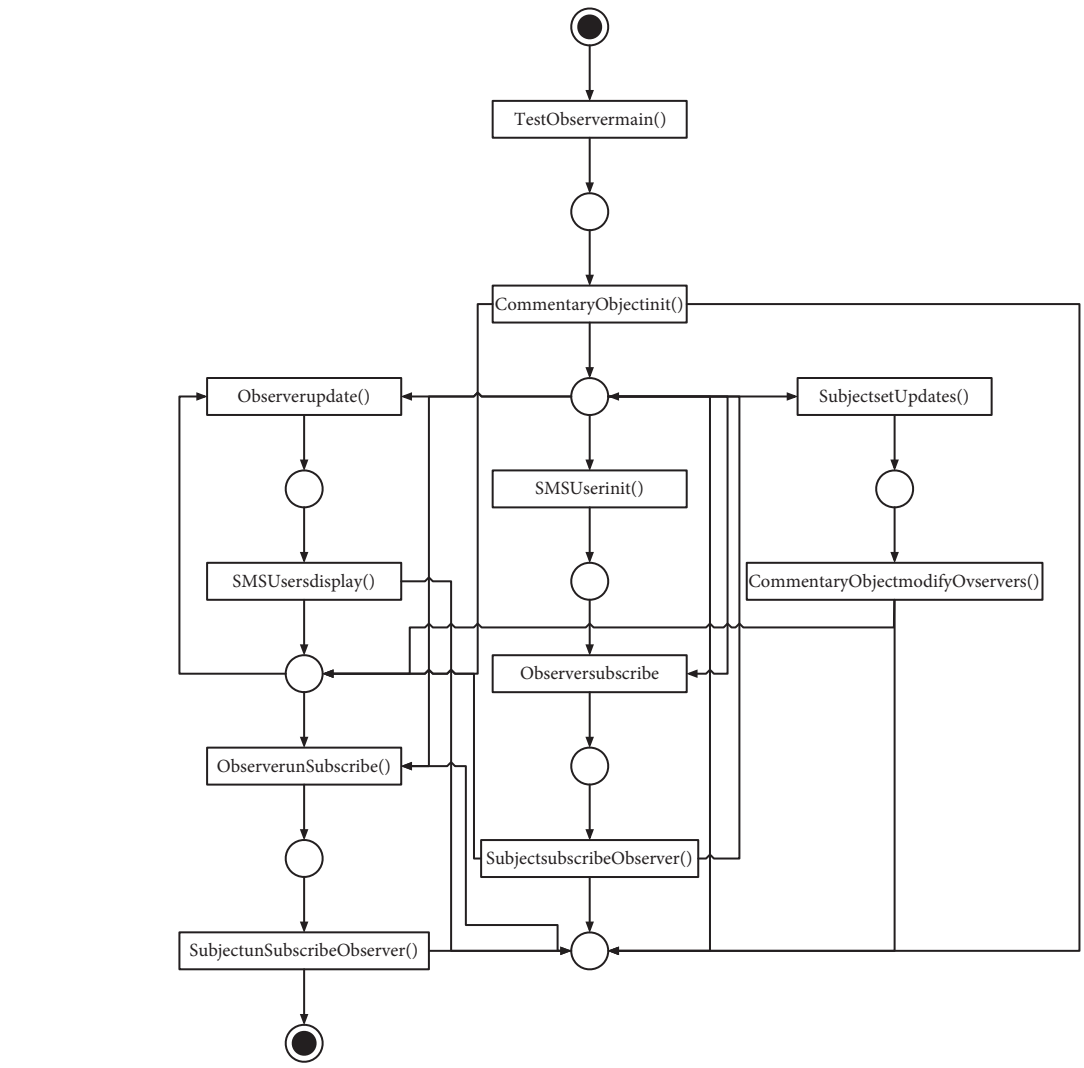
activities in the submodel, and *#total activities* is the number of all activities. After calculating the fitness of the two parts, the fitness value of the hierarchical process tree with components is obtained by multiplying the proportion of the two parts and adding them.

5.3.2. Simplicity. The complexity of the discovered software execution model is important for the comprehension and maintenance of software [42]. It has been shown that the size of a process model is the main indicator for perceived complexity and introduction of errors in process models [43]. A simple model can facilitate an easy understanding of execution behavior; then, better and more time-efficient maintenance can be achieved. Simplicity is measured according to [44] by comparing the size of the model to the number of activities in the log. The advantage of this method is that even if the model is unsound, its simplicity can be measured. The simplicity value defined in [44] is given by

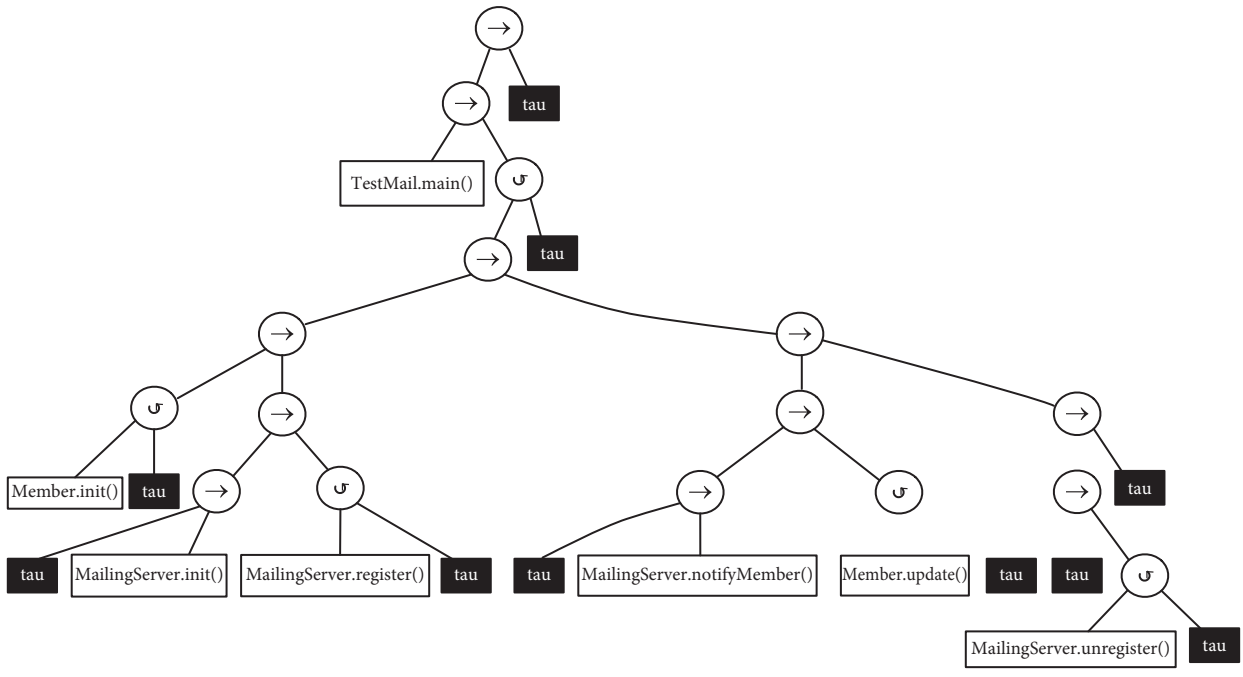
$$\text{simplicity} = 1 - \frac{\#\text{duplicate activities} + \#\text{missing activities}}{\#\text{nodes in model} + \#\text{total activities}}, \tag{3}$$

where *#duplicate activities* are the number of repeat times of activity in the process model. The missing activity is the activity that is recorded in the event log but not included in the model. These numbers are summed and normalized by the total number of nodes in the model and the activities in the event log.

The number of tau activities is also an important indicator of model complexity [45], tau activities exist in the process model but are not recorded in the event log. In some cases, the execution of a process needs to skip or repeat the current activity and jump to any previously executed activity, and tau activities are needed to play the role of



(a)



(b)

Figure 10: Continued.

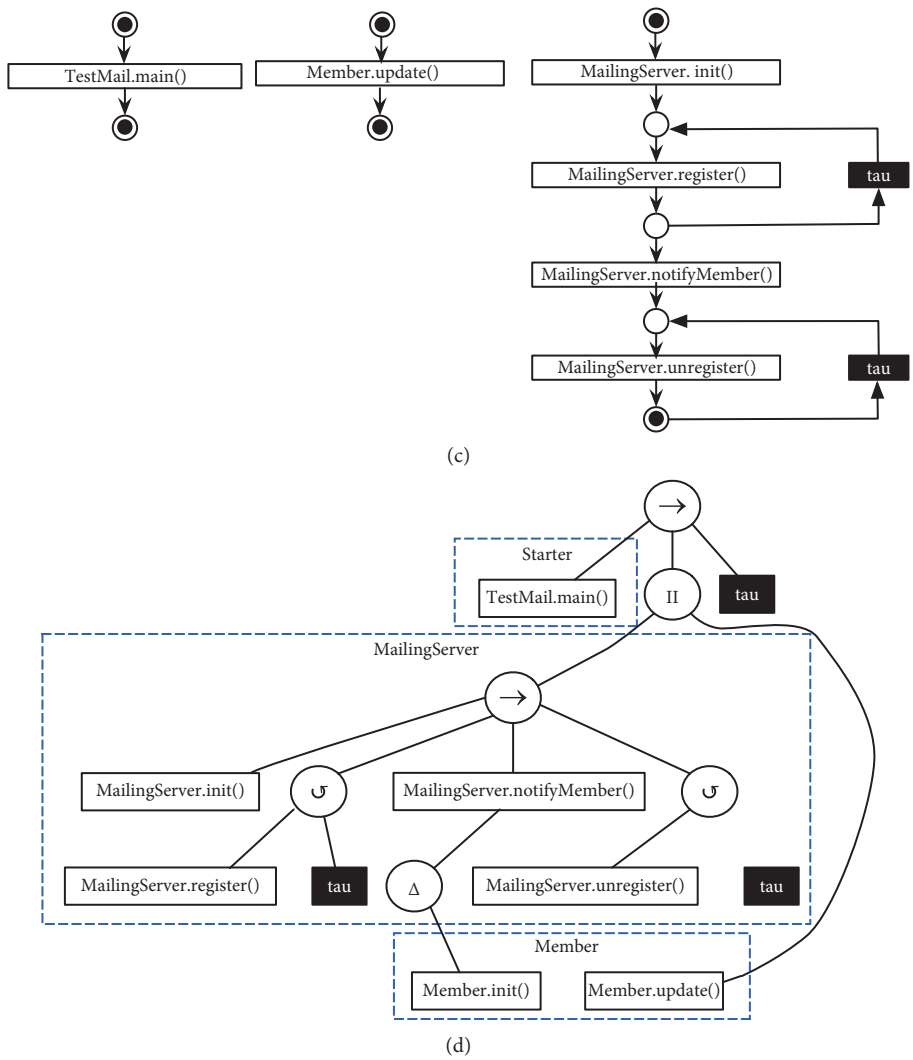


Figure 10: Models discovered by different algorithms on academic community data. (a) The alpha algorithm. (b) The IM algorithm. (c) The component-based method. (d) The proposed method.

"routing" in the process model. Tau activity is a human intervention that improves execution flexibility. To make the model as simple as possible, such human intervention should be minimum. Therefore, we add the consideration of tau activity based on (3), as shown in (4), where *#tau activities* is the number of tau activities that appear in the model.

$$\text{simplicity} = 1 - \frac{\#\text{duplicate activities} + \#\text{missing activities} + \#\text{tau activities}}{\#\text{nodes in model} + \#\text{total activities}}. \quad (4)$$

Table 7 lists the fitness and simplicity values of the six event log mining models. The bold value of each column means the maximum value of the column. Alpha, heuristics, ILP, component-based, and component-based (single-model) algorithms are unable to guarantee soundness, resulting in generating unsound models that cannot be measured by fitness, only the IM algorithm and our method can ensure that all the mining models are sound. Besides, heuristics, ILP, and fuzzy miner are unable to discover a model of the Apache dataset in the specified time (30 sec).

Compared with the proposed method, component-based method, and component-based (single model) method, the model generated by other algorithms had a relatively low fitness value because they could not discover the nested structure. In bookstore data, both component-based and the proposed method could correctly discover the flat and nested structures that exist in the event log, achieving perfect fitness. But if there are nested structures between different components, such as sport lobby and academic community data, the fitness of the model generated by component-based

Table 7: The quality of models discovered by different algorithms.

	Bookstore		Sport lobby		Academic community		JUnit 4.12		Apache		NASA CEV	
Algorithm	Fit	Sim	Fit	Sim	Fit	Sim	Fit	Sim	Fit	Sim	Fit	Sim
Alpha	0.796	0.923	−u	**0.961**	−u	0.714	−u	0.633	−u	0.610	0.870	0.529
Heuristics	0.796	0.923	−u	0.900	−u	**1**	−u	0.538	n/a	n/a	−u	0.510
ILP	0.692	0.928	−u	0.869	−u	0.875	n/a	n/a	n/a	n/a	−u	0.603
Fuzzy miner[1]	n/a	0.923	n/a	0.727	n/a	**1**	n/a	0.688	n/a	n/a	n/a	0.673
IM	0.881	0.577	0.976	0.285	0.946	0.700	0.770	0.470	0.557	0.355	0.863	0.690
Component-based[2]	**1**	**1**	0.923	0.786	0.823	0.85	0.651	0.928	−u	**0.625**	−u	**0.743**
Component-based (single model)	**1**	0.803	**1**	0.644	**1**	0.793	0.822	0.605	−u	0.572	−u	0.660
This paper	**1**	0.885	**1**	0.857	**1**	**1**	**0.86**	**0.953**	**0.786**	0.600	**0.890**	0.701

[1]Fuzzy miner is absent in fitness measure due to the lack of semantics for fuzzy models. [2]Component-based divides the event log to generate multiple submodels; the quality of the model is calculated by calculating the average value of each submodel. Fit: fitness; Sim: simplicity; U: unsound model; n/a: no model (time limited). The bold value of each column means the maximum value of the column.

Table 8: Running time of algorithms on different datasets (in milliseconds).

Algorithm	Bookstore	Sport lobby	Academic community	JUnit 4.12	Apache	NASA CEV
Alpha	**14**	**56**	**14**	**29**	**157**	**35**
Heuristics	236	98	66	537	_T	**410**
ILP	211	223	87	_T	_T	523
Fuzzy miner	906	826	783	329	_T	3692
IM	129	121	77	762	9654	965
Component-based	180	**94**	**69**	596	**7236**	2856
Component-based (single model)	220	104	89	890	10237	3485
This paper	**97**	109	75	**354**	8052	500

[T]Time limit exceeded 30 sec. The bold values of each column means the two largest values in each column.

was not ideal because component-based could not discover nested structures in different components, while the proposed method still achieve perfect fitness in these two datasets. The component-based (single-model) method can discover the nested structures within and between different components. Thus achieving perfect fitness values on bookstore, sport lobby, and academic community data.

The model discovered by the IM algorithm may contain several tau activities (see Figures 8(b), 9(b), and 10(b)). The models discovered by alpha algorithm, component-based (single model) method, and heuristics are complex in terms of the number of nodes or model structure, the alpha algorithm may also lead to duplicate activities, and the model discovered by ILP algorithm and the component-based method may contain several tau activities. All the above factors will reduce the simplicity. For the component-based method, the division of the event log can reduce the input complexity, and the division of the entire model into multiple submodels can improve the simplicity value of the model. However, it results in behavior loss between submodels, and the nested structures between different components could not be discovered, resulting in missing activities. The component-based (single model) method can discover the nested structures within and between different components, because the relationship between submodels can be found, and there will be some duplicate activities, which results in a more complex model. The proposed method may have tau activities; it can discover all nested structures, which can avoid some missing activities and duplicate activities. If nested relationships exist only within components, it is possible to achieve the highest simplicity compared with other methods (bookstore data).

5.4. Runtime Analysis. Table 8 shows the running time of algorithms on six datasets. The bold values of each column means the two largest values in each column. It can be seen from the picture that the alpha algorithm has advantages in running time. Heuristics, ILP, and fuzzy miner consume longer time compared to other algorithms and they are unable to discover the model of Apache dataset in the specified time (30 sec). All the above methods are unable to discover nested structures and ensure soundness. The component-based method is faster than the IM algorithm and the component-based (single model) in some datasets cause the division of the event log reducing the input complexity and speeding up the algorithm, but the component-based method results in behavior loss between sublogs. The proposed method has a better time performance than the IM algorithm, which causes the discovery of nest structures that can avoid some loops or parallel structures, thus speeding up the processing speed of the algorithm. Although the time performance of the proposed method is not as good as that of the component-based method on some datasets, our method avoids the loss of behavior between sublogs.

In summary, most process mining methods are unable to find nested structure, which leads to the low fitness degree of the model, and the resulting duplicate activities and missing activities will lead to a reduction in simplicity. The proposed method can effectively discover the nested structure behavior models with higher quality and comprehensibility through component information.

6. Conclusions

Existing process mining methods cannot generate models that can accurately reflect software execution behavior. Based on this problem, this paper proposes a component-based hierarchical software behavior model discovery method that identifies the hierarchical structure involved in software execution, generating models with high fitness and simplicity, and divides the mined models based on component information, which improves the comprehensibility of the model and discovers the internal execution and interaction behavior of components. The model can capture software execution behavior and provide useful insights into how software behaves; this helps to improve software usability and redesign, ultimately improving the usability and redesign of the software.

The proposed method can improve the model's comprehensibility by dividing the entire mining model into components, enabling it to better observe the interaction between components and internal running behavior. However, following the growth in software-scale and complicated structure, the execution of software process tends to become extremely complex. To further simplify the model, based on the observation that the interaction between components is completed through the interface, to identify the interface and divide the whole process model into multiple components and interfaces, then build the component model to present the internal interaction of components, and build the interface model to present the interaction between components will be our future work. This can further refine the process model and obtain a more accurate and understandable view that can reflect the behavior of software during the actual run-time execution.

Data Availability

The event logs of three real-life software systems have been converted to the IEEE XES format (http://www.xes-standard.org/. All source codes and event logs have been uploaded to the website https://github.com/TangYahui9596/Experiment-Data. And, the public datasets can be downloaded from [39–41].

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (grant no. 62002310); Yunnan Science and Technology Major Project (202002AD080002); Yunnan Provincial Natural Science Foundation Fundamental Research Project (202101AT070004 and 2019FB135); Yunnan Province Software Engineering Key Laboratory Open Fund Project (2020SE404); Yunnan University Data-Driven Software Engineering Provincial Science and Technology Innovation Team Project (2017HC012); Yunnan University "Dong Lu Young-Backbone Teacher" Training Program; Yunnan Philosophy and Social Science Youth Project (QN2020024); and the Postgraduate Project of Scientific Research Fund of Yunnan Provincial Department of Education (2021Y018).

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*, pp. 1–5, Association for Computing Machinery, Tallinn Estonia, August 2015.

[2] W. M. P. van der Aalst, *Process Mining - Data Science in Action*, Springer, Berlin, Germany, 2nd edition, 2016.

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

[4] C. Liu, "Discovery and quality evaluation of software component behavioral models," *IEEE Transactions on Automation Science and Engineering*, vol. 99, pp. 1–12, 2020.

[5] M. Leemans, W. M. P. V. D. Aalst, and M. G. J. V. D. Brand, "Recursion aware modeling and discovery for hierarchical software event log analysis," in *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 185–196, Campobasso, Italy, March 2018.

[6] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of Program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[7] 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*, pp. 248–257, IEEE, L'Aquila, Italy, September 2008.

[8] S. M. Ataee and Z. Bayram, "An improved abstract state machine based choreography specification and execution algorithm for semantic web services," *Scientific Programming*, vol. 2018, Article ID 4094951, 20 pages, 2018.

[9] A. Nanthaamornphong and A. Leatongkam, "Extended ForUML for automatic generation of UML sequence diagrams from object-oriented fortran," *Scientific Programming*, vol. 2019, Article ID 2542686, 22 pages, 2019.

[10] C. d. S. Garcia, A. Meincheim, E. R. Faria Junior et al., "Process mining techniques and applications - a systematic mapping study," *Expert Systems with Applications*, vol. 133, pp. 260–295, 2019.

[11] C. Liu, Q. Zeng, L. Cheng et al., "Privacy-preserving behavioral correctness verification of cross-organizational workflow with task synchronization patterns," *IEEE Transactions on Automation Science and Engineering*, vol. 18, no. 3 99, pp. 1037–1048, 2021.

[12] S. J. J. Leemans, D. Fahland, and W. M. P. V. D. Aalst, *Discovering Block-Structured Process Models from Event Logs-A Constructive Approach*, Springer, Berlin, Germany, 2013.

[13] J. C. A. M. Buijs, B. F. v. Dongen, and W. M. P. v. d. Aalst, "Quality dimensions in process discovery: the importance of fitness, precision, generalization and simplicity," *International Journal of Cooperative Information Systems*, vol. 23, no. 1, pp. 1–8, 2014.

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

[15] M. Leemans, d. A. Van, and M. P. Wil, "Process mining in software systems: discovering real-life business transactions and process models from distributed systems," in *Proceedings of the 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 44–53, Ottawa, Canada, September 2015.

[16] J. M. E. van der Werf, B. F. van Dongen, C. A. Hurkens et al., "Process discovery using integer linear programming," *Fundamenta Informaticae*, vol. 94, no. 3-4, pp. 387–412, 2009.

[17] Y. Tang, R. Zhu, T. li, F. Nan, M. Zheng, and Z. Ma, "Genetic process hybrid Mining based on Trace Clustering population," *Computer Integrated Manufacturing Systems*, vol. 26, no. 6, pp. 1510–1524, 1998.

[18] A. M. Lemos, C. C. Sabino, R. M. F. Lima et al., "Using process mining in software development process management: a case study," in *Proceedings of the IEEE International Conference on Systems*, pp. 1181–1186, Anchorage, AK, USA, October 2011.

[19] V. A. Rubin, A. A. Mitsyuk, I. A. Lomazova et al., "Process mining can Be applied to software too!" in *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering & Measurement*, pp. 51–57, Torino Italy, September 2014.

[20] S. Astromskis, A. Janes, and M. Mairegger, "A process mining approach to measure how users interact with software: an industrial case study," in *Proceedings of the International Conference on Software and Systems Process*, pp. 137–141, Tallinn Estonia, August 2015.

[21] W. M. P. van der Aalst, A. Kalenkova, V. Rubin, and E. Verbeek, "Process discovery using localized events," in *Application and Theory of Petri Nets and Concurrency*, pp. 287–308, Springer International Publishing, Berlin, Germany, 2015.

[22] C. Liu, "Automatic discovery of behavioral models from software execution data," *IEEE Transactions on Automation Science and Engineering*, vol. 15, no. 4, pp. 1897–1908, 2018.

[23] C. Liu, B. V. Dongen, N. Assy et al., "Component behavior discovery from software execution data," in *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, 2016*, Athens, Greece, December 2016.

[24] S. S. Emam and J. Miller, "Inferring extended probabilistic finite-state automaton models from software executions," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 1, 2018.

[25] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of UML sequence diagrams for distributed Java software," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 642–663, 2006.

[26] W. De Pauw, H. Lorenz David, M. Vlissides John et al., "Execution patterns in object-oriented visualization," in *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems*, p. 219, Santa Fe, New Mexico, April 1998.

[27] I. Beschastnikh, J. Abrahamson, Y. Brun et al., "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*, pp. 448–451, Szeged, Hungary, September 2011.

[28] M. J. H. Heule and S. Verwer, "Exact DFA identification using SAT solvers," in *Proceedings of the International Colloquium on Grammatical Inference*, pp. 66–79, Springer, Valencia, Spain, September 2010.

[29] A. Weijters and J. Ribeiro, "Flexible heuristics miner (FHM)," in *Proceedings of the 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pp. 310–317, IEEE, Paris, France, April 2011.

[30] C. W. Günther and W. M. Van Der Aalst, *Fuzzy Mining–Adaptive Process Simplification Based on Multi-Perspective Metrics*, Springer, Berlin, Germany, 2007.

[31] C. W. Gunther and H. Verbeek, "Xes-standard Definition," 2014, http://www.xes-standard.org/_media/xes/xesstandarddefinition-2.0.pdf.

[32] C. Srinivas, V. Radhakrishna, and C. V. G. Rao, "Clustering and classification of software component for efficient component retrieval and building component reuse libraries," *Procedia Computer Science*, vol. 31, pp. 1044–1050, 2014.

[33] H. Washizaki and Y. Fukazawa, "A technique for automatic component extraction from object-oriented programs by refactoring," *Science of Computer Programming*, vol. 56, no. 1-2, pp. 99–116, 2005.

[34] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review. E, Statistical, nonlinear, and soft matter physics*, vol. 69, no. 2, Article ID 026113, 2004.

[35] C. Liu, "Experimental data for "software data analytics: architectural model discovery and design pattern detection," in *Mathematics and Computer Science*Technische Universiteit Eindhoven, Eindhoven, Netherlands, 2019.

[36] R. Rotta and A. Noack, "Multilevel local search algorithms for modularity clustering," *Journal of Experimental Algorithmics*, vol. 16, no. 2(3), 2011.

[37] H. Verbeek, J. Buijs, B. Dongen et al., "ProM 6: The Process Mining Toolkit," 2010, https://www.promtools.org.

[38] A. van Hoorn, M. Rohr, W. Hasselbring et al., "Continuous Monitoring of Software Services: Design and Application of the Kieker Framework," Report TR-0921, University of Kiel, Kiel, Germany, 2009.

[39] M. M. Leemans, "JUnit 4.12 software event log," Eindhoven University of Technology, Eindhoven, Netherlands, 2016, https://data.4tu.nl/articles/dataset/JUnit_4_12_Software_Event_Log/12715829/1.

[40] M. M. Leemans, "Apache Commons Crypto 1.0.0 - Stream CbcNopad Unit Test Software Event Log," Eindhoven University of Technology, Eindhoven, Netherlands, 2017, https://data.4tu.nl/articles/dataset/Apache_Commons_Crypto_1_0_0_Stream_CbcNopad_Unit.

[41] M. M. Leemans, "NASA Crew Exploration Vehicle (CEV) Software Event Log," Eindhoven University of Technology, Eindhoven, Netherlands, 2017, https://data.4tu.nl/articles/dataset/NASA_Crew_Exploration_Vehicle_CEV_Software_Event_Log/12696995/1.

[42] C. Liu, Q. Zeng, L. Cheng, H. Duan, and J. Cheng, "Measuring similarity for data-aware business processes," *IEEE Transactions on Automation Science and Engineering*, vol. 99, pp. 1–13, 2021.
[43] R. Manfred and W. Barbara, *Enabling Flexibility in Process-Aware Information Systems*, Springer, Berlin, Germany, 2012.
[44] 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 *Proceedings of the OTM Confederated International Conferences" on the Move to Meaningful Internet Systems*, pp. 305–322, Springer, Rome, Italy, September 2012.
[45] L. Wen, J. Wang, W. M. P. van der Aalst, B. Huang, and J. Sun, "Mining process models with prime invisible tasks," *Data & Knowledge Engineering*, vol. 69, no. 10, pp. 999–1021, 2010.