PAPER

A Two-Layered Framework for the Discovery of Software Behavior: A Case Study

Cong LIU[†], Jianpeng ZHANG[††a)], Guangming LI[†††], *Nonmembers*, Shangce GAO[††††b)], *Member*, *and* Qingtian ZENG[†c)], *Nonmember*

SUMMARY During the execution of software, tremendous amounts of data can be recorded. By exploiting the execution data, one can discover behavioral models to describe the actual software execution. As a well-known open-source process mining toolkit, ProM integrates quantities of process mining techniques and enjoys a variety of applications in a broad range of areas. How to develop a better ProM software, both from user experience and software performance perspective, are of vital importance. To achieve this goal, we need to investigate the real execution behavior of ProM which can provide useful insights on its usage and how it responds to user operations. This paper aims to propose an effective approach to solve this problem. To this end, we first instrument existing ProM framework to capture execution logs without changing its architecture. Then a two-layered framework is introduced to support accurate ProM behavior discovery by characterizing both user interaction behavior and plug-in calling behavior separately. Next, detailed discovery techniques to obtain user interaction behavior model and plug-in calling behavior model are proposed. All proposed approaches have been implemented.

key words: *software behavior discovery, user behavior, plug-in calling behavior, process mining*

1. Introduction

Software systems form an integral part of the most complex artifacts built by humans, and we have become totally dependent on these complex software artifacts. Considering for example, communication, healthcare, education and government all increasingly rely on software. Modern enterprises continue to investing more and more in creation, maintenance and change of complex software systems. During the execution of software, execution data can be recorded. By fully exploiting the recorded data, one can discover behavioral models describing the actual execution of software.

Manuscript received January 18, 2017.
Manuscript revised July 20, 2017.
Manuscript publicized August 23, 2017.

[†]The authors are with Shandong Province Key Laboratory of Wisdom Mine Information Technology, Shandong University of Science and Technology, Qingdao 266590, China.
[††]The author is with National Digital Switching System Engineering Technological Research and Development Center, Zhengzhou 450000, China.
[†††]The author is with Science and Technology on Information Systems Engineering Laboratory, National University of Defense Technology, Changsha 410000, China.
[††††]The author is with University of Toyama, Toyama-shi, 930–8555 Japan.
a) E-mail: feilong0309@sina.com
b) E-mail: gaosc@eng.u-toyama.ac.jp
c) E-mail: qtzeng@163.com

DOI: 10.1587/transinf.2017EDP7027

Process mining [1] is an active research discipline that aims to extract process models from historic running logs. Researchers have proposed various process discovery techniques that take an event log as an input to produce a process model without using any priori information. Currently, the meaning of process mining is not limited to process discovery, but covers an even wider spectrum, such as conformance checking, model repair, performance predictions. For a comprehensive overview of process mining, one is referred to [1]. Most existing process mining techniques are implemented as plug-ins in the open-source process mining toolkit named ProM [2]. It takes event logs extracted from various information systems as inputs, and generates insightful analysis and verification results. Currently, ProM, as the *de facto* standard for process mining, contains more than 600 process mining plug-ins that have been successfully used in more than one hundred case studies to analyze business processes of organizations, such as banks, hospitals, municipalities, etc [1]. Moreover, worldwide research groups are being involved in contributing to its development and thousands of enterprises and organizations are downloading it. Given the popularity of ProM, how to develop a better ProM providing satisfying user experience is of vital importance. Obviously, as the first step to do so, one needs to understand the real ProM usage by characterizing its behaviors from history executing logs. Since our focus is on ProM software behavior understanding and not on its structure, process models plays a central role.

The scope of this work is to discover software behavior from its execution log by taking ProM as a case study. Here, software behavior refers to both user interaction behavior, i.e., how user interacts with software (ProM) and what operations are performed in what order, and the functional aspects of software (plug-in calling behavior for ProM). Existing ProM framework does not support the logging functionality, so our first step is to instrument it with event logging module without changing its original framework. XES, stands for *eXtensible Event Stream* [3], is a new logging format especially for process mining. We choose to log the execution information to XES formats, as it has the following four excellent features: (1) Simplicity. XES logs use the simplest way to represent information and it is easy to parse and generate; (2) Flexibility. XES standard is capable of capturing event logs from any background with regardless of its application domain; (3) Extensibility. It has transparent extension standard and maintains backward and forward compatibility. Similarly, it is also possible to extend for special requirements, e.g. for specific software process mining domains; and (4) Expressivity. Only those elements that can be identified in virtually any setting are explicitly defined by the standard, and additional information is deferred to optional attributes.

The remainder of this paper is organized as follows. Section 2 briefly introduces some related work. Section 3 presents how to instrument ProM framework to capture event logs without changing its original architecture. Then a two-layered framework is proposed to support the software behavior discovery by taking ProM as a case study in Sect. 4. Section 5 addresses its detailed behavior discovery approaches, including both plug-in calling relation discovery and user behavior discovery. Section 6 shows our experimental results. Finally, Sect. 7 concludes the paper.

2. Related Work

With the great flush of process mining techniques, it enables new forms of software runtime analysis, we call it *software process mining* [4]. Generally speaking, software process mining can be investigated at least in the following three perspectives: (1) mining software development processes, i.e., how software product is developed; (2) mining software systems running behavior, i.e., how software itself operates, understanding the functional aspects of a software system; and (3) mining user-software interaction process, i.e., how user interacts with software. The first aspect focuses on the analysis of software development lifecycle while the last two are about software runtime behavior discovery and analysis.

For the first perspective, Astromskis et al. [5] evaluated whether the creation of artifacts by a software development team violates certain CMMI practices previously defined rules. Lemos et al. used process mining to verify the conformance of a given software development process against an organizational standard [6]. Using process mining techniques, Poncin et al. addressed how software developers interact with software repositories [7], and Sebu and Ciocarlie [8] investigated the life cycles of entries (or artifacts) in issue tracking systems. For the other two perspectives, Rubin et al. [9] first discussed the application of some process mining algorithms to mining software and systems engineering processes from the information that is available in Software Configuration Management Systems. They argued that ProM could be used for analyzing and verifying some properties of these processes. In their recent work [10], they presented several process mining examples of different productive software systems used in the tourism domain. Based on their approach, Astromskis et al. [11] presented an industrial case to investigate how users interact with an enterprise resource planning software using process mining.

On one hand, our work belongs to a typical kind of software process mining case study, which emphasizes the real software behavior (ProM) analysis and typical user interaction behavior discovery. On the other hand, our method can be regarded as a special kind of hierarchical process mining, specifically tailored for the software domain. To overcome the "*spaghetti-like*" models which contain all details without any hierarchies, Gunther and van der Aalst [12] proposed the fuzzy mining approach. In this approach, activities and their relations are clustered and abstracted according to their importance to demonstrate different hierarchies or levels. However, the fuzzy miner does not have any semantic significance with respect to the domain, therefore it may suffer the risk of aggravating some irrelevant activities together to a cluster. Towards this limitation, Bose et al. [13] proposed hierarchical discovery approaches using a set of interrelated plug-ins in ProM to deal with fine-grained event logs and less structured process models. Different from the traditional fuzzy miner, the hierarchies are obtained through the automated discovery of pattern abstractions [14]. It is proved that the discovered patterns always have its specific domain semantics. Our work moves a step further by first discovering the domain related patterns (i.e., plug-in calling relations), and based on these discovered domain knowledge a hierarchical process model can be obtained to represent software behavior with more accuracy.

3. ProM Execution Log Collection

To discover the behavior of ProM, an event log that captures its real execution information is required. To create such an event log, this section introduces how to instrument the existing ProM framework with event recording function without changing its original architecture and runtime behavior.

3.1 Instrumentation

As existing ProM suit does not support event logging module, it is extremely rewarding to do this supplement. Similar to [15], we also attempt to manually instrument the source code to generate event log while software is running. Before adding the logging code manually to ProM framework, we need to define what we mean by an activity (event type) and how events are grouped, i.e., a case. Choosing an appropriate case notion is important as it can influence the results of our analysis. Considering for example, if we group events improperly without any repeat the discovered model may be extremely complex as it has to demonstrate every case. In our context, an event type corresponds to a user operation, essentially it is a plug-in execution. We assign a new case each time when the ProM is initialized, i.e., a case starts with the moment one launches the ProM software and ends when a user explicitly clicks the "exit" button. To capture runtime software execution logs, we do the following modification for the core ProM framework [†].

In this paper, we choose to store the captured event log using XES format [3], which is a standard format developed by the IEEE Task Force for logging events. Here we make the justification to choose XES briefly. It is an XML-based standard for recording event logs, and its purpose is

[†]https://svn.win.tue.nl/repos/prom/Framework/trunk/

to provide a generic format for the interchange of event log data among different applications. It is dedicated for process mining, i.e. the analysis of operational processes based on event logs. We collect name, event execution timestamp (start and end in milliseconds), originator and lifecycle information for each event.

A *LoggingEventsXES* class is added to perform the main log recording module. It is totally based on the *OpenXES* library, and contains three main functions, naming *saveCaseStart*(), *saveCaseEnd*() and *saveEventToLocal*(). The following code shows how to record the event that a user has started the ProM:

```
public void saveCaseStart(Date timestamp) {
XAttributeMap attMap = new XAttributeMapImpl();
XLogFunctions.putLiteral(attMap, "org:resource", "C.Liu");
XLogFunctions.putTimestamp(attMap, "time:timestamp", timestamp);
XLogFunctions.putLiteral(attMap, "concept:name", "End ProM");
XLogFunctions.putLiteral(attMap, "lifecycle:transition", "start");
XEvent event = new XEventImpl(attMap);
trace.add(event);}
```

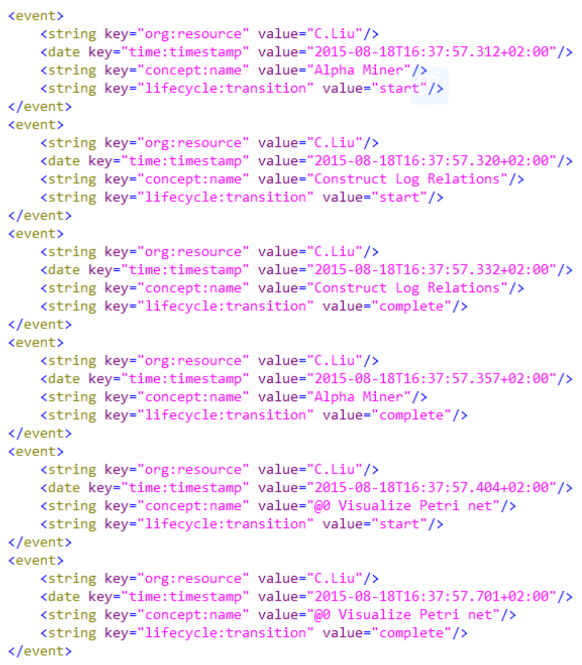
The *saveCaseStart*() function is added to *Boot.java* to record the start of ProM. Then the *saveCaseEnd*() function is added to class *UITopiaFrameLC*, which is an extension of *UITopiaFrame*, to record the end of ProM. Finally, we add *saveEventToLocal*() function to the *invoke*() of class *AbstractPluginDescriptor*, to capture start and complete events of each plug-in. Source code of the instrumented ProM framework with logging functionality is available online [†].

3.2 Simple Introduction of Recorded Event Logs

In this sub-section, we give a brief introduction of our recorded running logs. According to the XES standard, on the top level of an XES document there is one log object, which contains all event information that is related to one specific business process. One log contains an arbitrary number (may be empty) of traces, and each trace describes the execution of one specific instance or case. Every trace contains an arbitrary number (may be empty) of events. Each event represents an atomic activity that has been observed during the execution of a process. In our case, each activity refers to execution of a plugin, which is the basic function unit of ProM.

A screenshot of our recorded ProM execution running log fragment is shown in Fig. 1 which presents the typical information used for process mining, and its original event log is available in [16]. This log records the real usage of ProM with 69 cases, i.e., 69 times independent execution of ProM. The basic statistic information of our log is obtained using the log visualization option in ProM 6. Based on the visualization, we can see that (1) this event log contains 69 traces, 1033 events and 104 event classes; (2) on the right hand of the log visualization dashboard, the start and end date of the event log are August 14, 2015 and September 22, 2015 respectively; (3) the average event per case are 15 and the average event class per case are 10, but the distribution of the number of recorded events per case varies greatly.

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



```
<event>
    <string key="org:resource" value="C.Liu"/>
    <date key="time:timestamp" value="2015-08-18T16:37:57.312+02:00"/>
    <string key="concept:name" value="Alpha Miner"/>
    <string key="lifecycle:transition" value="start"/>
</event>
<event>
    <string key="org:resource" value="C.Liu"/>
    <date key="time:timestamp" value="2015-08-18T16:37:57.320+02:00"/>
    <string key="concept:name" value="Construct Log Relations"/>
    <string key="lifecycle:transition" value="start"/>
</event>
<event>
    <string key="org:resource" value="C.Liu"/>
    <date key="time:timestamp" value="2015-08-18T16:37:57.332+02:00"/>
    <string key="concept:name" value="Construct Log Relations"/>
    <string key="lifecycle:transition" value="complete"/>
</event>
<event>
    <string key="org:resource" value="C.Liu"/>
    <date key="time:timestamp" value="2015-08-18T16:37:57.357+02:00"/>
    <string key="concept:name" value="Alpha Miner"/>
    <string key="lifecycle:transition" value="complete"/>
</event>
<event>
    <string key="org:resource" value="C.Liu"/>
    <date key="time:timestamp" value="2015-08-18T16:37:57.404+02:00"/>
    <string key="concept:name" value="@0 Visualize Petri net"/>
    <string key="lifecycle:transition" value="start"/>
</event>
<event>
    <string key="org:resource" value="C.Liu"/>
    <date key="time:timestamp" value="2015-08-18T16:37:57.701+02:00"/>
    <string key="concept:name" value="@0 Visualize Petri net"/>
    <string key="lifecycle:transition" value="complete"/>
</event>
```

Fig. 1 Fragment of an XES File

According to Fig. 1, the running log of one event records its event name, resource, timestamp, and lifecycle information. For example, the event *Alpha Miner* starts at [2015-08-18T16:37:57.312] and ends at [2015-08-18T16:37:57.357], and it is operated by *C. Liu*.

Basic definitions and notions on event logs used for ProM behavior mining are recalled for self-completeness of the paper.

Definition 1: (**Event**) An *event* is defined as a 5-tuple $e = (AName, Cid, Org, StartTime, EndTime)$, where (1) $AName$ is the name or ID of the operation; (2) Cid refers the case which the operation runs in; (3) Org refers to the origination which execute this operation; (4) $StartTime$ is the start time of the operation; and (5) $EndTime$ is the end time of the operation.

Definition 2: (**Case, Log**) A *Case* is composed of a set of events and a *Log* is composed of a set of cases.

Definition 3: (**Activity Set**) Let $RLogs$ be an event log, for any $RCase \in RLogs, ActivitySet(RCase) = \{AName(e) | e \in RCase\}$ is the activity (or operation) set of $RCase$.

Basically, an event log contains recordings related to ProM execution. In addition, each event in the log needs to refer to a specific ProM execution instance, and often referred as a running case. Also, events are related to real ProM operations. In Fig. 1, events refer to activities (ProM operations) like *Alpha Miner*, *Construction Log Relations*, etc.

3.3 Petri Net

Petri net [17]–[28], is a type of bipartite graph with two

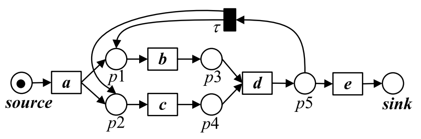


Fig. 2 A labeled petri net example

types of nodes, i.e., places and transitions. Transitions represent plug-ins and places are used to control the routing of transitions. To make our mining results more understandable, we add the formal definitions of Petri net and also presents an example Petri net to introduce its firing rule in the following.

Definition 4: (**Labeled Petri net**) A *Labeled Petri net* is a 4-tuple $PN = (P, T, F, l)$, satisfying: (1) P is a finite set of places and T is a finite set of transitions where $P \cap T = \emptyset$, $P \cup T \neq \emptyset$; (2) $F \subseteq (P \times T) \cup (T \times P)$ is set of directed arcs, called 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.

For each $x \in P \cup T$, ${}^{\bullet}x = \{y|(y, x) \in F\}$ is the preset of x and $x^{\bullet} = \{y|(x, y) \in F\}$ is the postset of x. A marking m of PN is a multiset of places, i.e., $m \in \mathbb{B}^{P}$, indicating how many tokens each place contains. (PN, m_0) is a *marked net* where m_0 is its *initial marking*. Figure 2 shows a marked Petri net example. $[source]$ is its initial marking. A transition $t \in T$ is *enabled* in marking $m \in \mathbb{B}^{P}$, denoted as $(PN, m)[t >$ if for each $p \in^{\bullet} t : m(p) \geq 1$. Considering the example Petri net with $m = [p_3, p_4]$, transition d is enabled, i.e., $(PN, m)[d >$. An enabled transition t may *fire* and results in a new marking m' with $m' = m \setminus^{\bullet} t \cup t^{\bullet}$, denoted by $(PN, m)[t > (PN, m')$. We have $(PN, [p_3, p_4])[d > (PN, [p_5])$ for the example Petri net.

4. A Two-Layered Framework for Software Behavior Discovery

In the previous section, we show how to extract ProM execution log by instrumentation. Next, we dive into the ProM behavior discovery approach by taking the recorded running log as inputs. To justify the necessity of our framework, we start with the limitations of existing approaches when analyzing real ProM execution log.

4.1 Why Existing Approaches Fail

First, we use two of the most widely used process mining techniques, *Fuzzy miner* and *Inductive miner*, to discover the behavior model of ProM usage by taking our event log as an input. The screenshots of their mining results are shown in Fig. 3 and Fig. 4 respectively.

By inspecting the discovered behavior models in details, we argue that they turn out to be less structured and *spaghetti-like* containing all kinds of details without distinguishing what is important (or relevant) and what is not from different angles. Specifically, they contain an excessive

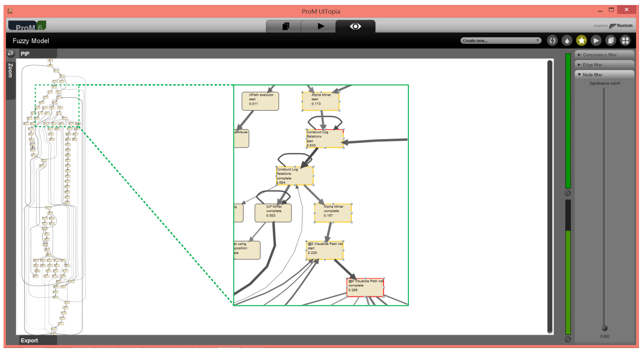


Fig. 3 Mining result using the fuzzy miner (with default setting)

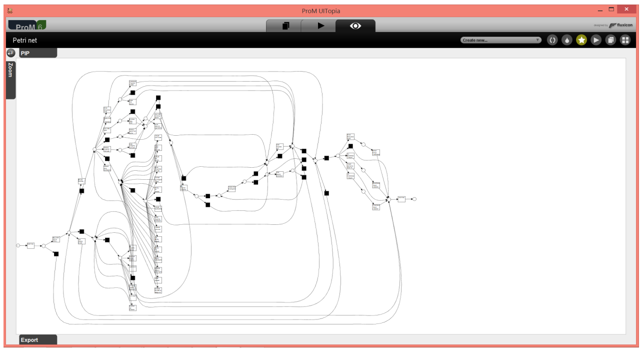


Fig. 4 Excerpt of mining result using the inductive miner (with default setting)

number of events which are not relevant with real user interaction behavior, i.e., they cannot demonstrate user behavior accurately. A ProM user may get confused by the resulting model, because some plug-ins are not explicitly triggered by them, but they are indeed recorded in the log. Considering for example, users may choose the *Alpha Miner* plug-in to discover a Petri net, therefore its start and end events are recorded as shown in Fig. 1. In addition, the start and end events of *Construct Log Relations* are also recorded. This is because that execution of *Alpha Miner* plug-in will implicitly call the *Construct Log Relations* plug-in during execution.

In general, this deficiency can be attributed to the following two assumptions of traditional process mining: (1) Any event type found in the log is assumed to have a corresponding ProM operation; and (2) All recorded events are recorded in the same level, and thus they are treated equally important. However, these assumptions are proved to be unreasonable. The recorded log contains not only the explicit user operations, but also automatically or implicitly calling action of some plug-ins. In other words, events on different levels are flattered into the same log without any discriminations. Motivated by the notion of fuzzy miner [12] and hierarchical process model discovery methods [4], [13], [14] which provide suitable abstractions of operational processes by clustering activities, we treat the ProM behavior with hierarchies. More specifically, the abstraction-based process mining idea by separating the original log into different lev-

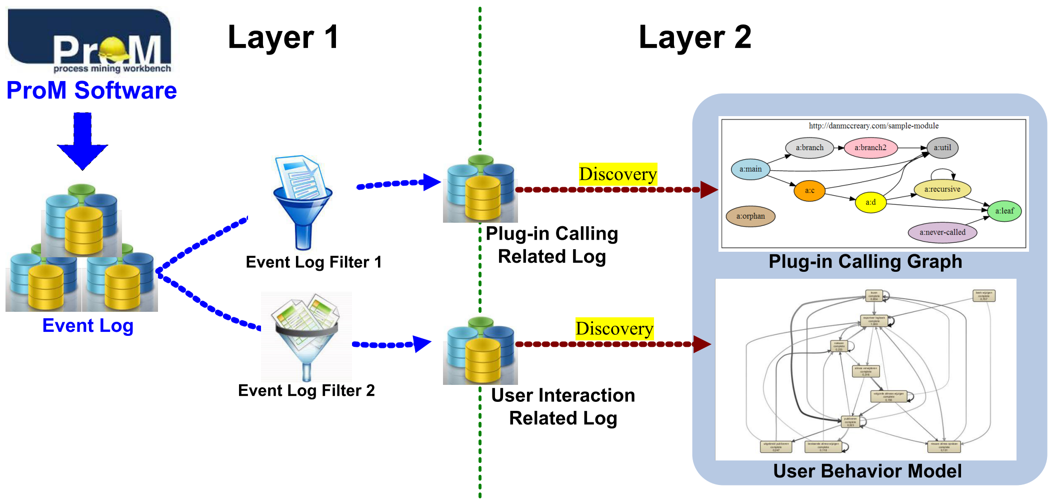


Fig. 5 A two layered discovery framework

els is adopted. A two-layered framework to discover ProM behavior on different abstractions is introduced in the next sub-section.

4.2 A Two-Layered Framework to Discover Software Behavior: A ProM Case Study

In last sub-section, we illustrate that existing mining techniques cannot do a satisfying work to discover real ProM behavior because they treat all recorded events equally. Actually, the log is a mixture of both user operation behaviors and plug-in automatically calling behaviors. In addition, some events are essentially related as their usage is confined to certain context or dedicated for some specific functionality. To achieve a better result, we first classify the log into two levels, naming the user interaction level and plug-in automatically calling level respectively. The former records the real and explicit user behavior while the latter implies the inherent plug-in calling relation of ProM. A detailed framework for ProM behavior discovery is illustrated in Fig. 5, which mainly includes the following two layers.

Layer 1: Recording ProM Running Logs and Preprocessing. While the ProM runs, actual execution information can be recorded in an XES formatted event logs. The event logs include information about event name, event originator, its start and end timestamps, etc. Because the logs are a mixture of both user interaction operations with ProM and plug-in automatically calling operations of ProM itself, we then classify them into **user interaction related running logs** and **plug-in calling related event logs**. The former contains all user interaction behavior related recordings and the latter involves only those plug-in calling information.

Layer 2: Discovering User Interaction Behavior and Plug-in Calling Behavior. After applying process mining techniques to user interaction related running logs, we can derive the real user operation process model which provides insights on the real usage of the ProM from a user perspective. Similarly, a plug-in calling graph can be discovered from the plug-in calling related event logs, which illustrates

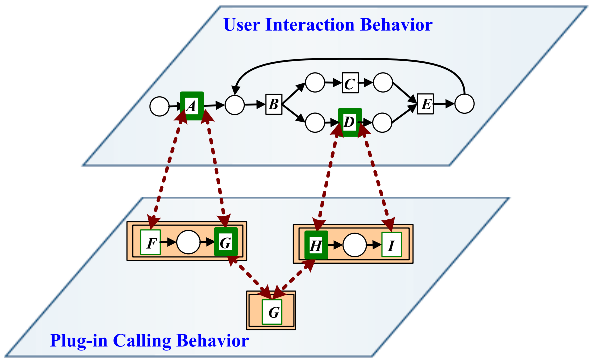


Fig. 6 Integrated ProM behavior representation

how ProM software really works and its running time plug-in calling relation. It is essentially a kind of ProM domain knowledge, which is kept in plug-in developer's mind.

Finally, to show the real ProM behavior one should combine the above two behavior models together with different hierarchies, and the final behavior model should look like the one shown in Fig. 6 where (1) the user operation behavior shows the real behavior of ProM from a user perspective; (2) plug-in calling behavior illustrates its run-time plug-in calling relations; and (3) the integrated behavior represents how ProM really works in real-life cases.

Different from the Fuzzy miner [12], the proposed two-layered behavior discovery approach is essentially domain-specific. We take the plug-calling relation as our dedicated domain knowledge which can be further used to realize ProM behavior abstraction. This way can overcome the deficiencies of aggravating some irrelevant activities together to a cluster. It is proved that the pattern-based hierarchical discovery technique [14] always has its specific domain semantics. Our approach moves a step further by directly discovering the domain related patterns (i.e., plug-in calling relations), and based on which a hierarchical behavior model is obtained. Some efforts in process mining try to address the semantic problems in the log specification [29].

However, it is required that domain experts have to come up with the ontologies to describe the domain concepts and relationships between them. Even though ontologies can assist in defining hierarchies of concepts over activities and providing abstractions, to ask domain experts to build this from scratch would extremely time-consuming and cost too much. From this perspective, one of the most interesting byproducts of our framework is the recognition of ProM domain knowledge, plug-in calling relation which is kept in plug-in developer's mind.

5. Comprehensive ProM Behavior Analysis

Section 4 introduced our two-layered framework to discover real ProM behaviors, including both plug-in calling behavior (can be treated as a kind of ProM domain knowledge which is kept in developers' mind) and user interaction behavior. This section details our behavior discovery approaches.

5.1 Plug-in Calling Behavior Discovery

The plug-in calling relation in Fig. 6 indicates that the execution of plug-in A needs to call plug-ins F and G, and the execution of plug-in G needs to call plug-in G. On the other hand, the execution of plug-in D calls plug-ins H and I, and the execution of plug-in H calls G. To illustrate the plug-in calling relation in a vivid manner, we propose to use plug-in calling graph which is motivated by the idea of call graph [30] in the programming area. More specifically, a plug-in calling graph is a directed graph that represents calling relationships between plug-ins. Each node represents a plug-in and each edge (M, N) indicates that plug-in M calls plug-in N. Formal definition of plug-in calling graph is given in the flowing.

Definition 5: (**Plug-in Calling Graph**) A plug-in calling graph is a tuple $PCG = (P, R)$ where P is a set of plug-ins, and R represents the calling relations of plug-ins.

The plug-in calling relations in Fig. 6 can be abstracted as $P = \{M, N, O, P\}$ and $R = \{(M, N), (M, O), (O, P)\}$, and drawn in a plug-in calling graph PCG_1 which should look like the one in Fig. 7.

To construct a plug-in calling graph directly from a ProM execution log, we need to investigate the dependency relations among each plug-ins. Here dependency relations among plug-ins can be inferred from their corresponding execution duration, i.e., the start and end timestamp of each plug-in.

Definition 6: (**Dependency Relation**) Let $RLogs$ be the running logs, and $\exists RCase_i \in RLogs$, $\forall A_i, A_j \in ActivitySet(RCase_i)$, A_i is dependent on A_j (or A_j is one of the dependent of A_i), denoted as $A_i \copyright A_j$ if $A_i.StartTime < A_j.StartTime$ and $A_i.EndTime < A_j.EndTime$ holds in $RLogs$.

Definition 7: (**Dependency Set**) $DependencySet(A_i)$ is defined as the dependency set of A_i, if $\forall A_j \in DependencySet(A_i)$, $A_i \copyright A_j$ holds.

The basic principle to define dependency relations among plug-ins is introduced as: if the recorded execution duration of plug-in A is completely covered by the other plug-in B, then B is dependent on A (or A is one of the dependent plug-ins of B). Considering for example, the start time of *Alpha Miner* is [2015-08-18T16:37:57.312] and its end time is [2015-08-18T16:37:57.357], while the start time of *Construct Log Relations* is [2015-08-18T16:37:57.320] and it ends at [2015-08-18T16:37:57.332], and their execution dependency relation is illustrated in Fig. 8. According to Definition 6, we know *Alpha Miner* is dependent on *Construct Log Relations*, denoted as $Alpha\ Miner \copyright Construct\ Log\ Relations$ or $Construct\ Log\ Relation \in DependencySet(Alpha\ Miner)$.

According to the basic notion of Definition 6, Algorithm 1 is given to discover plug-in dependency relations from an XES event log automatically.

Algorithm 1 To Obtain the Plug-in Dependency Relation from Event Logs.

```
Input: $RLogs$;
Output: $DependencySet$.
1: $DependencySet \leftarrow \emptyset$; $DependencySet(Activity_i) \leftarrow \emptyset$; $P \leftarrow \emptyset$; $R \leftarrow \emptyset$.
2: For each $RCase_i \in RLog$ Do
     For each $Activity_i \in RCase_i$ Do
       For each $Activity_j \in RCase_i$ Do
           If $Activity_i.StartTime < Activity_j.StartTime$ and $Activity_i.EndTime > Activity_j.EndTime$ then
               $DependencySet(Activity_i) \leftarrow DependencySet(Activity_i) \cup \{Activity_j\}$;
           End if
       End do
     End do
     $DependencySet \leftarrow DependencySet \cup DependencySet(Activity_i)$;
   End do
3: return $DependencySet$.
```

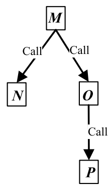


Fig. 7 Example of plug-in calling graph PCG_1

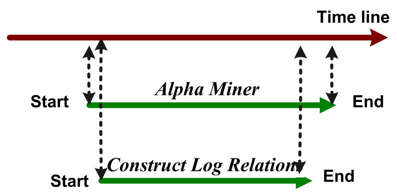


Fig. 8 Dependency relation example

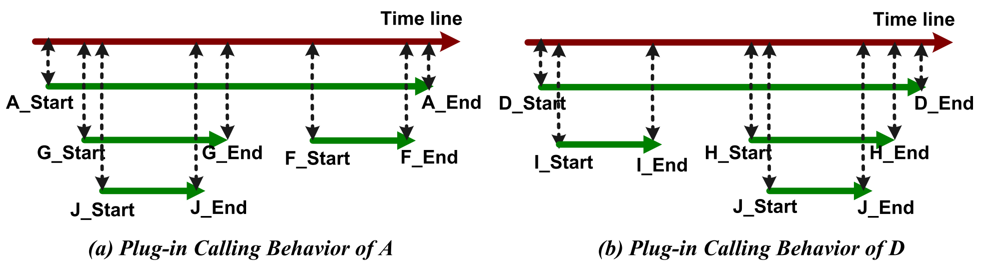


Fig. 9 Plug-in dependency relations

The complexity of Algorithm 1 is mainly determined by its second step whose complexity is $O(|RLog| * |RCase_i|^2)$ where $|RCase_i|$ is the number of activities in $|RCase_i|$ and $|RLog|$ is the number of cases in a log. Therefore, its complexity is $O(|RLog| * |RCase_i|^2)$.

Considering the plug-in dependency relations in Fig. 9, we have $DependencySet = \{DependencySet(A), DependencySet(F), DependencySet(G), DependencySet(J), DependencySet(D), DependencySet(H), DependencySet(I)\}$ where $DependencySet(A) = \{F, G\}$, $DependencySet(F) = \emptyset$, $DependencySet(G) = \{J\}$, $DependencySet(J) = \emptyset$, $DependencySet(D) = \{H, I\}$, and $DependencySet(H) = \{J\}$, $DependencySet(A) = \emptyset$.

Based on the plug-in dependency relations obtained from Algorithm 1, we can derive the plug-in calling graph easily using the following algorithm.

Algorithm 2 To Obtain the Plug-in Calling Graph from Dependency Relation.

```
Input: $DependencySet$;
Output: $PCG = (P, R)$.
1: $P \leftarrow \emptyset$; $R \leftarrow \emptyset$.
2: For each $DependencySet(Activity_i) \in DependencySet$ Do
     $P \leftarrow P \cup \{Activity_i\}$;
     For each $Activity_j \in DependencySet(Activity_i)$ Do
       $P \leftarrow P \cup \{Activity_j\}$;
       $R \leftarrow R \cup \{(Activity_i, Activity_j)\}$;
     End do
   End do
3: $PCG \leftarrow (P, R)$;
4: return $PCG$.
```

In Algorithm 2, the complexity of the second step is $O(|DependencySet| * |DependencySet(Activity_i)|)$. Because $|DependencySet| \geq |DependencySet(Activity_i)$, its complexity is $O(|DependencySet|^2)$. Hence, the complexity of Algorithm 2 is $O(|DependencySet|^2)$ where $|DependencySet|$ is the number of activities (plug-ins) in the log.

By taking the plug-in DependencySet in Fig. 9 as an input, we execute Algorithm 2 to discover its corresponding plug-in calling graph which should look like the one in Fig. 10. Following Definition 6, this plug-in calling graph can be formulated as $PCG_2 = (P_2, R_2)$ where $P_2 = \{A, D, F, G, H, I, G\}$ and $R_2 = \{(A, F), (A, G), (D, H), (D, I), (G, J), (H, J)\}$.

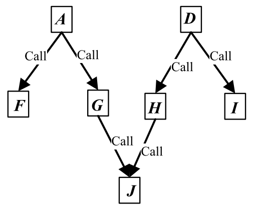


Fig. 10 Example of plug-in calling graph of *Fig.* 9

5.2 User Behavior Discovery

To discover the user behavior model, we first introduce how to filter the event log to containing only explicit user interaction information. The plug-in calling information needs to be removed from the original event log. By taking the original event log as an input, we use the following algorithm to generate user interaction related event log following the next algorithm.

Algorithm 3 To Obtain the User Interaction Related Event Log.

```
Input: $RLogs$ and $DependencySet$;
Output: $URLogs$.
1: $URLogs \leftarrow \emptyset$; $TempCase \leftarrow \emptyset$; $P \leftarrow \emptyset$; $R \leftarrow \emptyset$.
2: For each $RCase_i \in RLog$ Do
     For each $Activity_i \in RCase_i$ Do
       $TempCase \leftarrow RCase_i$;
       For each $Activity_j \in RCase_i$ Do
           If $Activity_i.StartTime < Activity_j.StartTime$ and
   $Activity_i.EndTime > Activity_j.EndTime$ then
             $TempCase \leftarrow TempCase - \{Activity_j\}$;
         End if
       End do
     End do
     $URLogs \leftarrow URLogs \cup TempCase$;
   End do
3: return $URLogs$.
```

The complexity of Algorithm 3 is mainly determined by its second step whose complexity is $O(|RLog| * |RCase_i|^2)$ where $|RCase_i|$ is the number of activities in $|RCase_i|$ and $|RLog|$ is the number of cases in a log. Therefore, its complexity is $O(|RLog| * |RCase_i|^2)$.

By running Algorithm 3, we obtain the user interaction related event log, based on which the user behavior can be discovered directly. It is worth noting that the behavior discovery process is not limited to some specific mining algorithms and all existing miners that are capable of dealing with basic process patterns can be applied.

6. Experimental Results

To show the effectiveness and applicability of the proposed approach in discovering software behavior, we developed the corresponding functions as ProM plug-ins. The first plug-in, *Mine the plug-in Calling Graph* plug-in is developed to discover the plug-in calling graph from the original event log. It implements Algorithms 1- 2, which first discovers the plug-in dependency relations and then converts them to plug-in calling graph. By taking our recorded running logs in [16] as an input, we run this plug-in and the excerpt of the plug-in calling graph mined from the original log should look like the one shown in Fig. 11. Considering for example, the plug-in *construct log relation* is called by both *Alpha Miner* and *ILP Miner*. Plug-in calling relations that are kept in developers mind can be regarded as a kind of ProM domain knowledge. It can be used for (1) plugin developers to check if the real execution is performed as programmed; and (2) new plugin developers to refer to the realization principle of some existing plugins which have the same functionality.

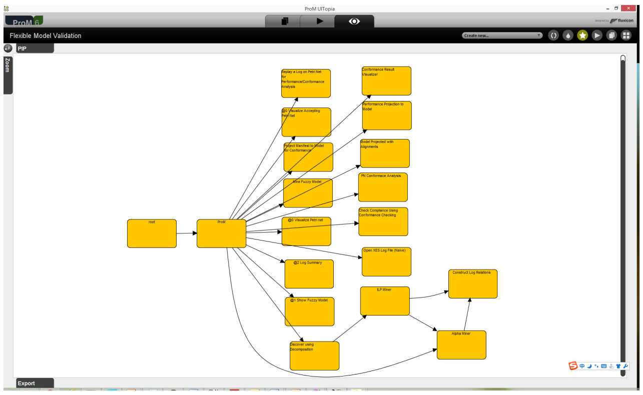

ProM UITopia
Flexible Model Validation
Create new...
PIP
Zoom
Export
root
ProM
Replay a Log on Petri Net for Performance/Conformance Analysis
@0 Visualize Accepting Petri Net
Project Manifest to Model for Conformance
Mine Fuzzy Model
@0 Visualize Petri net
@2 Log Summary
@1 Show Fuzzy Model
Discover using Decomposition
Conformance Result Visualizer
Performance Projection to Model
Model Projected with Alignments
PN Conformace Analysis
Check Compliance Using Conformance Checking
Open XES Log File (Naive)
ILP Miner
Alpha Miner
Construct Log Relations

Fig. 11 Excerpt of plug-in calling graph using *mine the plug-in calling graph* plug-in

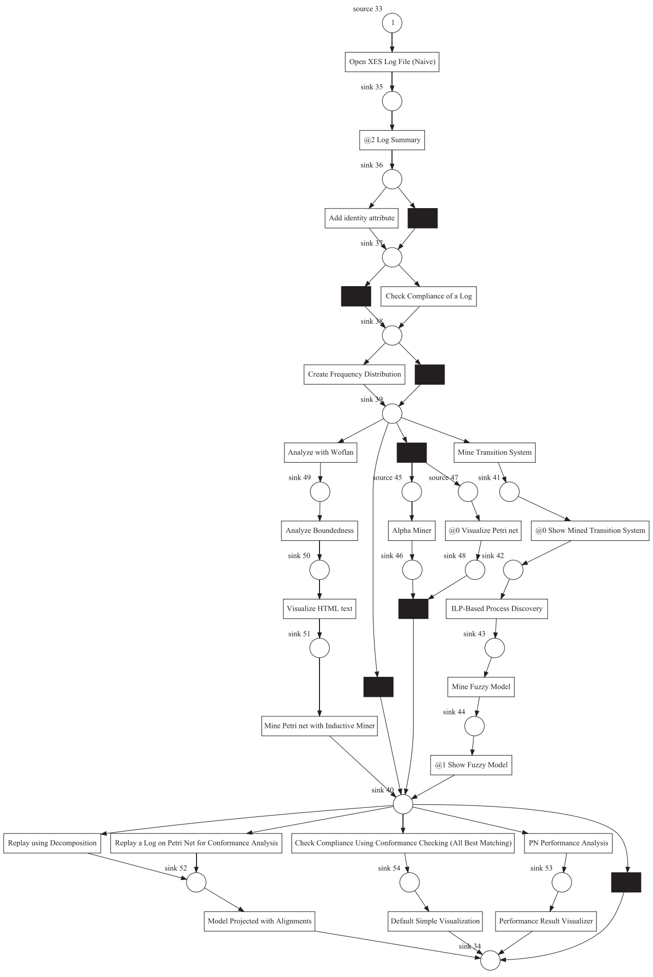

source 33
Open XES Log File (Naive)
sink 35
@2 Log Summary
sink 36
Add identity attribute
sink 37
Check Compliance of a Log
sink 38
Create Frequency Distribution
sink 39
Analyze with Woflan
Mine Transition System
sink 49
source 45
source 47
sink 41
Analyze Boundedness
Alpha Miner
@0 Visualize Petri net
@0 Show Mined Transition System
sink 50
sink 46
sink 48
sink 42
Visualize HTML text
ILP-Based Process Discovery
sink 51
sink 43
Mine Fuzzy Model
sink 44
Mine Petri net with Inductive Miner
@1 Show Fuzzy Model
sink 40
Replay using Decomposition
Replay a Log on Petri Net for Conformance Analysis
Check Compliance Using Conformance Checking (All Best Matching)
PN Performance Analysis
sink 52
sink 54
sink 53
Model Projected with Alignments
Default Simple Visualization
Performance Result Visualizer
sink 34

Fig. 12 Mining result of filtered log using *inductive miner* (with default setting)

The second plug-in, named as *Plug-in Behavior Filter* is developed to obtain explicit user interaction information from the event log, i.e., plug-in calling information needs to be removed from the original event log. It is the implementation of Algorithm 3. By taking our recorded running log in [16] as an input, we run the *Plug-in Behavior Filter* plug-in. The user interaction related running log can be obtained and this event log contains 69 traces, 534 events and 78 event classes.

Using the obtained user interaction related running log, user behavior model can be discovered. As mentioned previously, the behavior discovery is not limited to some specific mining algorithms and all existing miners that are capable of dealing with basic process patterns can be applied. To give a clear demonstration of effectiveness and applicability of the proposed two-layered approach, we use the *Inductive Miner* which is considered as the state-of-the-art process discovery algorithm. The screenshot of the mining result (in terms of Petri net) is shown in Fig. 12.

By carefully looking at the discovered behavior model, we can see that it turns out to be well structured and contains only user interaction related behaviors, i.e., it reveals user behavior accurately. The implicitly calling plug-in information is excluded, and a ProM user may know quite well of its own operation behavior by the resulting model.

7. Conclusion

During software execution, actual execution information can be recorded as an event log. To character the ProM behavior in an accurate manner, we project the original log to a user interaction related event log and a ProM plug-in calling related event log. After applying process mining techniques to both logs, we can derive ProM plug-in calling behavior model and user interaction behavior model. These models provide insights on the real usage of the software (or how ProM really functions), and can enable usability improvement and further ProM redesign. Our scope focuses on software behavior analysis by using the well-known process mining toolkit ProM as a case study. The proposed framework is readily applicable on real-time ProM behavior monitoring to gain insights on both explicit user operating behavior and implicit ProM plug-in calling behavior. Note that the current technique is only valid when the actions are executed in foreground processes and background (e.g., fork, asynchronous) processes are not supported yet.

Currently, we manually instrument ProM framework to capture the plug-in level recordings which are very high-level. To discovery the execution behavior of the software (ProM) with more accuracy, one may prefer to capture the method-level event log. Based on method-level event log, it is possible to discover detailed software runtime behavior. However, the low-level recordings cannot reveal high-level operations (like user behavior) directly. Therefore, how to bridge the gap between low-level logs and high-level user operations is a pressing work. In addition, some more practical questions can be answered based on the discovery results. For example, what is the average/minimum/maximum execution of each case? To answer such questions, we need to enrich the user behavior model time factors. Similarly, to answer questions like which plug-in has the highest calling frequency?, we need to enrich the plug-in calling graph with frequency information according to the log.

Acknowledgments

This research was partially supported by the JSPS KAKENHI Grant Number JP17K12751, the National Key Research and Development Program of China under Grant 2016YFB0800101, the National Natural Science Foundation of China under Grant 61472229, the Foundation for Innovative Research Groups of the National Natural Science Foundation of China under Grant 61521003, the Science and Technology Development Fund of Shandong Province of China under Grant 2016ZDJS02A11 and ZR2017MF027, the Taishan Scholar Climbing Program of Shandong Province, and the SDUST Research Fund under Grant 2015TDJH102.

References

[1] W. van der Aalst, Process mining: discovery, conformance and enhancement of business processes, Springer Science & Business Media, 2011.

[2] 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," Applications and Theory of Petri Nets 2005, pp.444–454, Springer, 2005.

[3] H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst, "Xes, xesame, and ProM 6," Information Systems Evolution, pp.60–75, Springer, 2011.

[4] C. Liu, S. Wang, S. Gao, F. Zhang, and J. Cheng, "User Behavior Discovery from Low-level Software Execution Logs," IEEJ Trans. Electrical and Electronic Engineering, pp.1–9, in press, 2018.

[5] S. Astromskis, A. Janes, A. Sillitti, and G. Succi, "Implementing organization-wide gemba using non-invasive process mining," Cutter IT Journal, vol.26, no.4, pp.32–39, 2013.

[6] A.M. Lemos, C.C. Sabino, R.M.F. Lima, and C.A.L. Oliveira, "Using process mining in software development process management: A case study," 2011 IEEE International Conference on Syst. Man Cybern. (SMC), pp.1181–1186, IEEE, 2011.

[7] W. Poncin, A. Serebrenik, and M. van den Brand, "Process mining software repositories," 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), pp.5–14, IEEE, 2011.

[8] M.L. Sebu and H. Ciocarlie, "Applied process mining in software development," 2014 IEEE 9th International Symposium on Applied Computational Intelligence and Informatics (SACI), pp.55–60, IEEE, 2014.

[9] 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," Software Process Dynamics and Agility, pp.169–181, Springer, 2007.

[10] V.A. Rubin, A.A. Mitsyuk, I.A. Lomazova, and W.M.P. van der Aalst, "Process mining can be applied to software too!," Proc. 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, p.57, ACM, 2014.

[11] S. Astromskis, A. Janes, and M. Mairegger, "A process mining approach to measure how users interact with software: an industrial case study," Proc. 2015 International Conference on Software and System Process, pp.137–141, ACM, 2015.

[12] C. Günther and W. van der Aalst, "Fuzzy mining–adaptive process simplification based on multi-perspective metrics," International Conference on Business Process Management, pp.328–343, Springer, 2007.

[13] R.P.J.C. Bose, E.H.M.W. Verbeek, and W.M.P. van der Aalst, "Discovering hierarchical process models using ProM," IS Olympics: Information Systems in a Diverse World, pp.33–48, Springer, 2012.

[14] R.P.J.C. Bose and W.M.P. van der Aalst, "Abstractions in process mining: A taxonomy of patterns," Business Process Management, pp.159–175, Springer, 2009.

[15] R. Pérez-Castillo, B. Weber, I.G.-R. de Guzman, and M. Piattini, "Process mining through dynamic analysis for modernising legacy systems," IET software, vol.5, no.3, pp.304–319, 2011.

[16] Original ProM event log:. https://www.dropbox.com/s/scy2miujgifu0xe/Original%20ProM%20Execution%20Log-Plug-in%20Level.xes?dl=0.

[17] C. Liu, Q. Zeng, J. Zou, F. Lu, and Q. Wu, "Invariant decomposition conditions for petri nets based on the index of transitions," Information Technology Journal, vol.11, no.7, pp.768–774, 2012.

[18] C. Liu, H. Duan, Q. Zeng, M. Zhou, F. Lu, and J. Cheng, "Towards comprehensive support for privacy preservation cross-organization business process mining," IEEE Trans. Services Computing, 2016.

[19] C. Liu and F. Zhang, "Petri net based modeling and correctness verification of collaborative emergency response processes," Cybernetics and Information Technologies, vol.16, no.3, pp.122–136, 2016.

[20] C. Liu, J. Cheng, W. Yirui, and S. Gao, "Time performance optimization and resource conflicts resolution for multiple project management," IEICE Trans. Inf. & Syst., vol.E99-D, no.3, pp.650–660, 2016.

[21] C. Liu, Q. Zeng, H. Duan, M. Zhou, F. Lu, and J. Cheng, "E-net modeling and analysis of emergency response processes constrained by resources and uncertain durations," IEEE Trans. Syst. Man Cybern.: Systems, vol.45, no.1, pp.84–96, 2015.

[22] C. Liu, Q. Zeng, H. Duan, and F. Lu, "Petri net based behavior description of cross-organization workflow with synchronous interaction pattern," International Workshop on Process-Aware Systems, pp.1–10, Springer, 2014.

[23] C. Liu, Q. Zeng, and H. Duan, "Formulating the data-flow modeling and verification for workflow: A petri net based approach," Int. J. Science and Engineering Applications, vol.3, pp.107–112, 2014.

[24] Q. Zeng, C. Liu, and H. Duan, "Resource conflict detection and removal strategy for nondeterministic emergency response processes using petri nets," Enterprise Information Systems, vol.10, no.7, pp.729–750, 2016.

[25] Q. Li, Y. Deng, C. Liu, Q. Zeng, and Y. Lu, "Modeling and analysis of subway fire emergency response: an empirical study," Safety science, vol.84, pp.171–180, 2016.

[26] J. Cheng, C. Liu, M. Zhou, Q. Zeng, and A. Ylä-Jääski, "Automatic composition of semantic web services based on fuzzy predicate petri

nets," IEEE Trans. Automation Science and Engineering, vol.12, no.2, pp.680–689, 2015.

[27] Q. Zeng, F. Lu, C. Liu, H. Duan, and C. Zhou, "Modeling and verification for cross-department collaborative business processes using extended petri nets," IEEE Trans. Syst. Man Cybern.: Systems, vol.45, no.2, pp.349–362, 2015.

[28] Q.-T. Zeng, F.-M. Lu, C. Liu, and D.-C. Meng, "Modeling and analysis for cross-organizational emergency response systems using petri nets," Chin. J. Comput, vol.36, no.11, pp.2290–2302, 2013.

[29] A.K.A. de Medeiros, W. van der Aalst, and C. Pedrinaci, "Semantic process mining tools: core building blocks," 2008.

[30] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," ACM SIGPLAN Notices, vol.32, no.10, pp.108–124, 1997.

Cong Liu received his M.S. degree in computer science and technology from Shandong University of Science and Technology, Qingdao, China in 2015. His research interests are in the areas of Business Process Management, Process mining, Petri nets and Software runtime analysis.

Jianpeng Zhang received B.S. degree in Electronic Information Engineering from Hebei University of Technology, TianJian, China, in 2010. In 2013, he earned the M.E. degree in Communication and Information System at NDSC, and now he is a Ph.D. student in Eindhoven University of Technology. His research interests include big data processing, machine learning and graph database.

Guangming Li received his B.S. degree in electrical engineering from Harbin Institute of Technology, Harbin, China, in 2012 and M.S. degree in management science and engineering from National University of Defense Technology, Changsha, China in 2014. His research interests are in the areas of business process modeling, process deviation detection and Petri net.

Shangce Gao received his Ph.D. degree in innovative life science from University of Toyama, Toyama, Japan in 2011. He is currently an Associate Professor with the Faculty of Engineering, University of Toyama, Japan. From 2011 to 2012, he was an associate research fellow with the Key Laboratory of Embedded System and Service Computing, Ministry of Education, Tongji University, Shanghai, China. From 2012 to 2014, he was an associate professor with the College of Information Sciences and Technology, Donghua University, Shanghai, China. His main research interests include mobile computing, nature-inspired technologies, and neural networks for optimization and real-world applications. He was a recipient of the Shanghai Rising-Star Scientist award, the Chen-Guang Scholar of Shanghai award, the Outstanding Academic Performance Award of IEICE, and the Outstanding Academic Achievement Award of IPSJ. He is a Senior Member of IEEE.

Qingtian Zeng received the Ph.D. degree in computer software and theory from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2005. He is currently a Professor with Shandong University of Science and Technology, Qingdao, China. His research interests are in the areas of Petri nets, process mining, and knowledge management.