Component Interface Identification and Behavioral Model Discovery from Software Execution Data

Cong Liu[1], Boudewijn van Dongen[1], Nour Assy[1], Wil M.P van der Aalst[2,1]
[1]Eindhoven University of Technology, 5600MB Eindhoven, The Netherlands.
[2]RWTH Aachen University, 52056 Aachen, Germany.
{c.liu.3,b.f.v.dongen,n.assy}@tue.nl,wvdaalst@pads.rwth-aachen.de

ABSTRACT

Restructuring an object-oriented software system into a component-based one allows for a better understanding of the system and facilitates its future maintenance. A component-based architecture structures a software system in terms of its components and interactions where each component refers to a set of classes. To represent the architectural interaction, each component provides a set of interfaces. Existing interface identification approaches are mostly structure-oriented rather than function-oriented. In this paper, we propose an approach to identify interfaces of a component according to the functional interaction information that is recorded in the software execution data. In addition, we also discover the contract (represented as a behavioral model) for each identified interface by using process mining techniques to help understand how each interface actually works. All proposed approaches have been implemented in the open source process mining toolkit *ProM*. Using a set of software execution data containing more than 650.000 method calls generated from three software systems, we evaluate our approach against three existing interface identification approaches. The empirical evaluation demonstrates that our approach can discover more functionally consistent interfaces which facilitate the reconstruction of architectural models with higher quality.

KEYWORDS

Interface identification, Interface behavioral model, Process mining, Software execution data, Empirical Evaluation

ACM Reference Format:
Cong Liu[1], Boudewijn van Dongen[1], Nour Assy[1], Wil M.P van der Aalst[2,1]. 2018. Component Interface Identification and Behavioral Model Discovery from Software Execution Data. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension, May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196338

1 INTRODUCTION

Software architecture reconstruction aims to abstract, identify, and present higher level views from lower level data to help understanding software [13], [23]. The inputs can be source code, development documents, software execution data, etc., and the outputs are different architectural views, which normally include components and interfaces. For object-oriented software systems, a component is extracted as a set of classes, which provides a number of functions to other components. When a component interacts with another, it requires one or more functions that are provided by the other component. These functions are implemented as sets of methods which are usually organized into interfaces.

Given a component, the identification of its interfaces can be achieved by clustering its methods [1], [19]. Existing interface identification approaches can be classified into structure-oriented and function-oriented. Structure-oriented approaches [1] cannot identify interfaces that represent typical functions of a component if these functions are not reflected in the internal structure. Function-oriented approaches [19] address this limitation by analyzing the interaction among components recorded in software execution data. However, existing function-oriented approaches define interfaces in a coarse-grained way which often results in complex interfaces with low functional consistency and high inter-dependency. In addition, the interface definition is rigid and cannot be adapted according to the user's desired level of inspection (i.e., fine-grained versus coarse-grained).

This paper considers the problem of identifying interfaces of components that are clearly defined, i.e., our starting point is a component configuration. This component configuration can either be given or discovered from execution data [10]. Given the component configuration of a piece of software, we identify interfaces of a component by taking as input software execution data. This provides rich information on how software components are executed and are interacting with each other. The analysis of all interactions of a component allows us to discover different functions that are represented as different sets of methods and extracted as interfaces of the component. Concretely, we first propose an approach to identify interfaces by grouping methods that are called by the same method. Compared to [19], this approach identifies interfaces in a fine-grained way. To improve the flexibility of the approach, we introduce a threshold that allows merging the identified interfaces to a desired level of granularity. In addition, when an interface is used by a component, the execution of its methods should follow a specific *contract*. This contract defines the behavior of the interface by explicitly specifying in which order the methods should be or can be invoked. After identifying the method set of an interface, we discover a behavioral model using process mining techniques [24] to help understand the contract.

The rest of this paper is organized as follows. Section 2 presents a review of the related work. Section 3 details the interface identification and behavioral model discovery approach. Section 4 introduces a ProM plugin implementing the approach. Section 5 presents experimental evaluation of the proposed approach. Finally, Section 6 concludes the paper and presents further research directions.

2 RELATED WORK

2.1 Software Architecture Reconstruction

Software architecture plays a pivotal role in software understanding, reuse, evolution, maintenance, etc. [3], [23]. It contains components and interfaces. For software systems that are implemented by object-oriented technology, a component is composed of a set of classes, and an interface is composed of a set of methods. Various clustering-based techniques, e.g., [2], [4], [15], [20], [22], [28] are proposed to identify components from source code or execution data according to different criteria such as coupling and cohesion. As for interface identification, *Simon et al.* [1] propose to identify interfaces by grouping methods of the same class, i.e., one interface for each class if this class has methods that are used by other components. Hence, this approach defines interfaces based on the internal structure of components. *Seriai et al.* [19] give an approach to identify the interfaces of a component by grouping methods that are called by another component. Methods that are used by the same component(s) are grouped as an interface. Hence, this approach leads to different components using a single interface regardless of the functions they need.

2.2 Software Process Mining

With the development of process mining [9], [21], [24], [30] on the one hand, and the growing availability of software execution data on the other hand, a new form of software analytics comes into reach, i.e., applying process mining techniques to analyze software execution data. This inter-disciplinary research area is called *Software Process Mining (SPM)* [10], [18], and aims to analyze software execution data from a process-oriented perspective. One of the first papers addressing *SPM* is [25]. *Leemans and van der Aalst* [7] discover and analyze the operational processes of software systems using process mining techniques. However, the hierarchical structure of software is not fully considered. By taking full consideration of component-based architecture and hierarchical structure of a software system, [10] proposes to discover a hierarchical behavioral model for each component. The discovered component model describes the software behavior from the perspective of individual components. However, this work neglects the interfaces that each component provides to other components.

3 COMPONENT INTERFACE DISCOVERY

This section details our main approach to discover interfaces and their behavioral models from software execution data. Our approach requires two inputs: (1) software execution data which are recorded by instrumenting the execution of software; and (2) a component configuration that specifies how classes are grouped to form components. In *Section* 3.1, we give an overview of the identification and discovery approach. In *Sections* 3.2-3.3, we present the details of the discovery techniques.

Table 1: An Example of Software Execution Data

ID	(Callee) Method	(Callee) Object	Caller Method	Caller Object
m_1	Class2.init	5746e7cc	MainClass.main	mainclass
m_2	Class3.init	9aixyao6	MainClass.main	mainclass
m_3	Class3.invoke	9aixyao6	MainClass.main	mainclass
m_4	Class1.init	3b7359cb	Class3.invoke	9aixyao6
m_5	Class2.setClass1	5746e7cc	Class1.init	3b7359cb
m_6	Class1.perform	3b7359cb	Class3.invoke	9aixyao6
m_7	Class2.work	5746e7cc	Class1.perform	3b7359cb
m_8	Class3.invoke	9aixyao6	MainClass.main	mainclass
m_9	Class1.init	614b152d	Class3.invoke	9aixyao6
m_{10}	Class2.setClass1	5746e7cc	Class1.init	614b152d
m_{11}	Class1.perform	614b152d	Class3.invoke	9aixyao6
m_{12}	class2.work	5746e7cc	Class1.perform	614b152d
m_{13}	MainClass.main	mainclass	–	–

3.1 Overview of the Approach

Generally speaking, our interface identification and discovery approach consists of the following two phases:

Phase 1: Interface Identification. Starting from the software execution data and component configuration, we propose to identify a set of interfaces for each component in two steps.

- ***(a) Candidate Interface Identification.*** For each component, we identify a set of candidate interfaces by grouping methods with respect to their caller methods; and
- ***(b) Similar Interface Candidate Merge.*** To solve the method duplication among candidate interfaces within the same component, we merge similar candidates in a way such that the overlap of shared methods among interfaces is limited.

Phase 2: Interface Contract Discovery. For each identified interface, we discover its contract (a behavioral model) to represent the actual behavior using process mining techniques in two steps.

- ***(a) Interface Event Log Construction.*** To enable the discovery of interface behavioral model using process mining techniques, we obtain the event log from the software execution data for each identified interface.
- ***(b) Interface Contract Discovery.*** For each interface, we discover a behavioral model by process mining techniques.

3.2 Phase 1: Interface Identification

A *method call* is the basic unit of software execution data [11], [12], [6]. Technically, it is recorded at a specific time and contains the fully quantified names of the caller and callee. Let $\mathcal{U}_M$ be the method call universe, $\mathcal{U}_N$ be the method universe, $\mathcal{U}_C$ be the class universe, $\mathcal{U}_{CO}$ be the component universe, and $\mathcal{U}_O$ be the object universe where objects are instances of classes. For any $m \in \mathcal{U}_M$, $\widehat{m} \in \mathcal{U}_N$ is the method of which m is an instance. For any $o \in \mathcal{U}_O$, $\widehat{o} \in \mathcal{U}_C$ is the class of o. $\eta : \mathcal{U}_M \to \mathcal{U}_O$ is a mapping from method calls to its objects such that for each method call $m \in \mathcal{U}_M$, $\eta(m)$ is the object containing the instance of the method $\widehat{m}$. $c : \mathcal{U}_M \to \mathcal{U}_M \cup \{\perp\}$ is the calling relation among method calls. For $m \in \mathcal{U}_M$, if $c(m) = \perp$, then $\widehat{m}$ is the *main* method. For any $m_i, m_j \in \mathcal{U}_M$, $c(m_i) = m_j$ means that m_i is called by m_j, and we name m_i as the *callee* and m_j as the *caller*.

Definition 1. ***(Software Execution Data)*** *Software execution data is a totally ordered set of method calls, i.e., $SD \subseteq \mathcal{U}_M$.*

Considering the execution data in Table 1 which is generated by one execution of an example piece of software. Each row corresponds to a method call, we use *ID* to uniquely represent each method call recording, e.g., m_1 is the first method call.

A component is composed of a set of classes. Let $COM \subseteq \mathscr{U}_{CO}$ be the component set of a piece of software iff for any $C_i, C_j \in COM$, $C_i \cap C_j = \emptyset$ or $C_i = C_j$. For the example program, we assume that it has three components, i.e., $C_1 = \{Class1, Class2\}$, $C_2 = \{Class3\}$ and $C_M = \{MainClass\}$. Note that we assume that *COM* to be known. However, [2] presents an approach to discover *COM* from software execution data.

From software execution data, we obtain component execution data by grouping method calls according to the components.

Definition 2. ***(Component Execution Data)*** *Let SD be software execution data and $C \in COM$ be a component. The execution data of C are $SD_C = \{m \in SD | \widehat{\eta(m)} \in C\}$, i.e., all method calls referring to objects which are instances of classes in C.*

According to Definition 2, the execution data of components C_1, C_2 and C_M are $SD_{C1} = \{m_1, m_4, m_5, m_6, m_7, m_9, m_{10}, m_{11}, m_{12}\}$, $SD_{C2} = \{m_2, m_3, m_8\}$, and $SD_{CM} = \{m_{13}\}$.

Generally speaking, interfaces are based on the functional aspects of a component. The set of all functional aspects provided by a component is represented by its top-level (or exposed) method set. Top-level methods are methods that are exposed to the outside and are directly called from other components.

Definition 3. ***(Top-level Method Set of a Component)*** *Let $C \in COM$ be a component and SD_C be its execution data. The top-level method set of C is $MS_C = \{n \in \mathscr{U}_N | \exists m \in SD_C : \widehat{m} = n \wedge \widehat{\eta(c(m))} \notin C\}$, i.e., all methods that are called by methods of other components.*

The top-level method set of component C_1 is $MS_{C1} =$ *{Class2.init, Class1.init, Class1.perform}.*

When a component interacts with another, it means that it requires one or more functions that are provided by the latter. A trivial solution to interface discovery is to claim that the set of top-level methods is the interface of a component. This means that other components can only use this single interface regardless of the functionality they use. This definition of interface is too general and hinders the understanding of the relationships between components. Instead, the analysis of all interactions of a component allows us to define different functional interfaces of a component each represented by a set of methods of one or more classes. In the following, we first identify candidate interfaces and then merge them into interfaces. More specifically, interfaces are obtained by grouping top-level methods with respect to their caller methods, i.e., methods of the same candidate interface are called by the same method. The rationale behind this is that if a set of methods is called by the same method of another component, they probably share the same function or need to work together to fulfill a function. We name these methods *functionally related methods.*

From the top-level method set of a component, we obtain candidate interfaces by grouping methods according to the caller methods, i.e., methods in the same candidate interface are called by the same method and methods in different candidate interfaces are called by different methods.

Table 2: Another Example of Software Execution Data

ID	(Callee) Method	(Callee) Object	Caller Method	Caller Object
m'_1	C1.init	613lcaixy	M.main	mainclass
m'_2	C1.a	613lcaixy	M.main	mainclass
m'_3	C1.b	613lcaixy	M.main	mainclass
m'_4	C2.init	1314xyalc	M.main	mainclass
m'_5	C2.d	1314xyalc	M.main	mainclass
m'_6	C1.init	520xinglc	C2.d	1314xyalc
m'_7	C1.a	520xinglc	C2.d	1314xyalc

Definition 4. ***(Candidate Interface Set of a Component)*** *Let $C \in COM$ be a software component, SD_C be its execution data and MS_C be top-level method set. $\tilde{I}^n_C \subseteq MS_C$ is a candidate interface as called by $n \in \mathscr{U}_N$ if and only if $\tilde{I}^n_C = \{x \in MS_C | \exists m \in SD_C : \widehat{m} = x \wedge \widehat{c(m)} = n\}$. $\widetilde{IS}_C = \{\tilde{I}^n_C \subseteq MS_C | n \in \mathscr{U}_N\}$ is the candidate interface set of C.*

According to Definition 4, the candidate interface set of component C_1 is $\widetilde{IS}_{C1} = \{\tilde{I}^1_{C1}, \tilde{I}^2_{C1}\}$ where we have $\tilde{I}^1_{C1} = \{Class2.init\}$ and $\tilde{I}^2_{C1} = \{Class1.init, Class1.perform\}$.

The candidate interfaces obtained by only considering their caller methods may cause duplication, i.e., some methods may be included in different candidates. To demonstrate this point, we give another software example. This software has three classes such that each forms an independent component, i.e., *Starter* with class *M*, *COM₁* with class *C1* and *COM₂* with class *C2*. An excerpt of its execution data are given in Table 2, based on which we can see (1) *C1.init*, *C1.a* and *C1.b* are invoked by *M.main* from *Starter*; and (2) *C1.init* and *C1.a* are invoked by *C2.m* from *COM₂*. Therefore, we obtain two candidate interfaces $\tilde{I}^1_E = \{C1.init, C1.a, C1.b\}$ and $\tilde{I}^2_E = \{C1.init, C1.a\}$ for COM_1. In this case, methods *C1.init* and *C1.a* are duplicated in both of them. To solve the duplication problem, we propose a heuristic to merge similar candidate interfaces in a way such that the shared methods among interfaces are limited to a reasonable range. The similarity between two interfaces are measured using the *Jaccard distance* based on their method sets.

Definition 5. ***(Similarity of Candidate Interface)*** *Let $\widetilde{IS}_C$ be a set of the candidate interfaces. For any $\tilde{I}^1_C, \tilde{I}^2_C \in \widetilde{IS}_C$, $Sim(\tilde{I}^1_C, \tilde{I}^2_C) = |\tilde{I}^1_C \cap \tilde{I}^2_C| / |\tilde{I}^1_C \cup \tilde{I}^2_C|$ is defined as the similarity between $\tilde{I}^1_C$ and $\tilde{I}^2_C$.*

For example, the similarity between candidate interfaces $\tilde{I}^1_{C1}$ and $\tilde{I}^2_{C1}$ of C_1 is $Sim(\tilde{I}^1_{C1}, \tilde{I}^2_{C1}) = 0$ as they do not share any methods. Similarly, the similarity between candidate interfaces $\tilde{I}^1_E$ and $\tilde{I}^2_E$ can be computed as: $Sim(\tilde{I}^1_E, \tilde{I}^2_E) = 0.67$.

Definition 6. ***(Interface Set of a Component)*** *Let $C \in COM$ be a software component and δ be a threshold satisfying $0 < \delta \leq 1$. $IS_C \subseteq \mathcal{P}(MS_C)$ is the interface set of C such that: for any $I^1_C, I^2_C \in IS_C$, we have $Sim(I^1_C, I^2_C) < \delta$.*

Considering the candidate interfaces $\tilde{I}^1_{C1}$ and $\tilde{I}^2_{C1}$ of C_1, if we set $\delta = 0.5$ we can get the interface set of component C_1 as $IS_{C1} = \{I^1_{C1}, I^2_{C1}\}$. This is because $Sim(\tilde{I}^1_{C1}, \tilde{I}^2_{C1}) < \delta$, therefore the interfaces are not merged. If we consider the candidate interfaces $\tilde{I}^1_E$ and $\tilde{I}^2_E$ and set $\delta = 0.5$, we will get one merged interface with methods *C1.init*, *C1.a* and *C1.b* as $Sim(\tilde{I}^1_E, \tilde{I}^2_E) > \delta$. Differently, if we set $\delta = 0.8$ which is greater than the similarity value, $\tilde{I}^1_E$ and $\tilde{I}^2_E$ are returned without merging and methods *C1.init* and *C1.a* are duplicated in both interfaces.

One extreme case is that all candidate interfaces of a component are merged to a single interface according to the similarity threshold. This interface may become too general and whatever functions are needed it will be used. To address this extreme case, users should be very careful with the threshold setting.

3.3 Phase 2: Interface Contract Discovery

Normally, when an interface is called by a component, the execution of its methods should follow a specific contract. This contract defines the behavior of the interface by explicitly specifying in which order its methods should be invoked. In this section, we discover a behavioral model to represent its contract for each identified interface using process mining techniques [24]. To do so, an event log is made for each interface and process mining is used to discover behavior.

From component execution data and interface set, we obtain interface execution data.

Definition 7. ***(Interface Execution Data)*** *Let $C \in COM$ be a software component, IS_C be the interface set of C and SD_C be the software execution data of C. For each $I \in IS_C$, $SD_I = \{m \in SD_C | \widehat{m} \in I \wedge \widehat{\eta(c(m))} \notin C\}$ is the execution data of interface I, i.e., all method calls referring to methods of interface I and are called by methods of other components.*

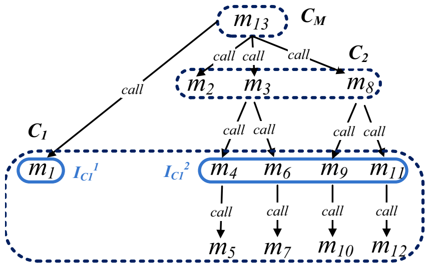


Figure 1: Invocation Structure

Fig. 1 shows the invocation structure of the execution data in Table 1. The execution data of interface I_{C1}^2 is $\{m_4, m_6, m_9, m_{11}\}$.

The basis of process mining is an event log. Within an event log, events are typically grouped by cases and totally ordered. To obtain an event log from software execution data for each interface, the notion of case is required to relate different method calls which correspond to events. The choice of case determines the scope of the discovered model, therefore we identify independent interface instances as cases.

An *interface instance* represents one concrete occurrence/instantiation of the interface contract and the software execution data of an interface may contain one or more interleaved interface instances. For object-oriented software, an interface instance involves a number of interacting objects. To facilitate identification of interface instances, we (1) construct an *Object Interaction* graph for the interface execution data; and (2) detect its *weakly connected components* which represent different interface instances.

Definition 8. ***(Object Interaction Graph)*** *$G_I = (V_I, R_I)$ is the Object Interaction (OI) graph of SD_I such that (1) $V_I = \{\eta(m) | m \in SD_I\}$; and (2) $R_I = \{(o_1, o_2) \in V_I \times V_I | \exists m \in SD_I : \eta(c(m)) = o_1 \wedge \eta(m) = o_2\}$.*

An *OI* graph is a directed graph where vertices represent objects and edges represent interactions among them. The interaction relation among objects is obtained from the calling relation among method calls. After constructing an *OI* graph, the set of vertices included in each *weakly connected component* corresponds to an interface instance. A *weakly connected component* of a directed graph is a maximal group of vertices that are mutually reachable by violating the edge directions [16].

Definition 9. ***(Interface Instance)*** *Let SD_I be the interface execution data and $G_I = (V_I, R_I)$ be its OI graph. $\Phi_I \subseteq \mathcal{P}(V_I)$ is the interface instance set of SD_I such that: (1) $\bigcup_{\varphi \in \Phi_I} \varphi = V_I$; (2) $\forall_{\varphi^1, \varphi^2 \in \Phi_I} \varphi^1 \cap \varphi^2 = \emptyset \wedge R_I \cap ((\varphi^1 \times \varphi^2) \cup (\varphi^2 \times \varphi^1)) = \emptyset$; and (3) $\nexists \Phi_I' \subseteq \mathcal{P}(V_I) : |\Phi_I'| > |\Phi_I|$.*

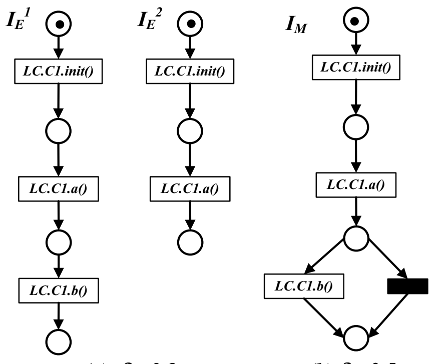
By applying Definition 9 to the software interface execution data, we identify all instances of this interface. Based on the identified interface instances, we construct software cases by grouping method calls in the execution data.

Definition 10. ***(Software Case)*** *Let SD_I be the interface execution data and Φ_I be its corresponding interface instance set. The case function $\xi : SD_I \rightarrow \Phi_I$ is a mapping from a method call (software event) to its corresponding instance (case). For any $m \in SD_I$, we have $\eta(m) \in \xi(m)$.*

According to Definition 10, each software case refers to an interface instance and is composed of a sequence of software events (referring to method calls). After constructing an event log for each interface, we discover a model to describe its actual behavior. Theoretically, we can use any existing process discovery approaches. In this paper, we use *Inductive Miner (IM)* [8] which is known as a state-of-the art process discovery approach. By taking the event log of an interface as input, *IM* returns a behavioral model in terms of workflow net [26], [29], [14].

Considering the example software execution data in Table 2. If we set the similarity threshold $\delta = 0.8$, the discovered interface behavioral models are shown in Fig. 2 (a). We get two interfaces (I_E^1 and I_E^2) and methods *C1.init* and *C1.a* are duplicated in both of them. This is because $Sim(I_E^1, I_E^2) < \delta$, therefore the interfaces are not merged. Differently, if we set the similarity threshold $\delta = 0.5$, the discovered interface behavioral model is shown in Fig. 2 (b). We get only one interface I_M by merging method sets of I_E^1 and I_E^2 as $Sim(I_E^1, I_E^2) > \delta$. For the interface behavioral model in Fig. 2 (b), rectangles represent methods and arcs together with circles control the execution flow of methods. The place (circle) with a token in it (black dot) means the starting point of the model. The behavior can be interpreted as: it starts with the initialization of *C1* and followed by *C1.a*. Then, *C1.b* can be executed or skipped.

According to the interface behavioral models in Fig. 2, we can see that (1) I_E^1 and I_E^2 each provide an independent function, say *function1* and *function2*; and (2) I_M is a general interface that provides functions of both I_E^1 and I_E^2. Even though the duplication and the number of interfaces are reduced when using I_M as the interface of component $C1$, the function that I_M represents is more general. Whenever *function1* or *function2* are required by other components, interface I_M will be used.



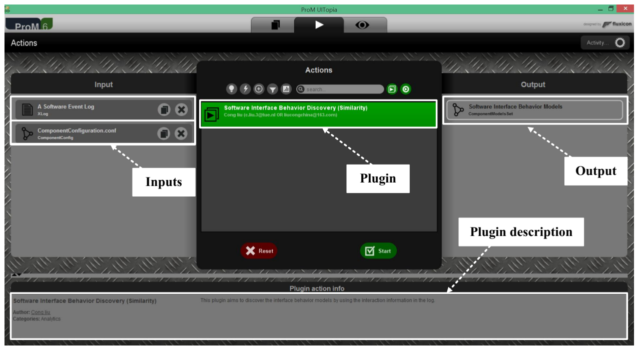
(a) $\delta = 0.8$ *(b)* $\delta = 0.5$

Figure 2: Interface Behavioral Models

4 IMPLEMENTATION IN PROM

The open-source (Pro)cess (M)ining framework *ProM 6* [27] has been developed as a pluggable took-kit for process mining and related topics. It can be extended by adding plug-ins. Currently, more than 1600 plug-ins are available. The framework is available.[1]

Our approach has been implemented as an integrated plug-in, called *Software Interface Behavior Discovery*, in the *ProM 6* package.[2] It takes (1) software execution data as an event log, and (2) a component configuration file that describes which class belongs to which component as input, and returns a set of interface behavioral models for each component. All discovered interfaces in the following experimental evaluation section are based on this tool. A snapshot of the tool is shown in Fig. 3.



This plugin aims to discover the interface behavior models by using the interaction information in the log.

Figure 3: Snapshot of the Tool

5 EMPIRICAL EVALUATION

In this section, we first introduce the subject software systems and the execution data. Then, we define the research questions as well as the quality metrics. Next, we evaluate the impact of different similarity thresholds on the quality of identified interfaces and show the advantages and applicability of our approach by comparing it with existing approaches.

Table 3: Basic Statistics of the Software Execution Data

Software	#Package	#Class	#Method	#Method Call
Google Sheet	7	22	57	3180
JGraphx	10	88	712	267627
JHotDraw	5	102	572	392164

5.1 Subject Software Systems and Execution Data

For our experiments, we use three software systems from different application domains. *Google Sheets* (formerly called the Google Spreadsheets API) is an open-source library that lets users develop client applications that read and modify worksheets and data in Google Sheets. *JGraphx* is an open-source family of libraries that provide features aimed at applications that display interactive diagrams and graphs. *JHotDraw* is a Java GUI framework for technical and structured 2D Graphics.

Table 3 shows some descriptive statistics for these software systems, including the number of packages/classes/methods that are loaded during execution and number of method calls analyzed.

5.2 Quality Metrics

To evaluate the quality of the identified interfaces, we define the following three different quality criteria.

- **Functional consistency of the identified interfaces.** Does the approach **correctly group** functionally related methods to the same interface? Does the approach **correctly separate** functionally unrelated methods to different interfaces? It is desirable to identify interfaces that neither wrongly group nor wrongly separate methods.
- **Functional inter-dependency of the identified interfaces.** Does the approach identify interfaces that are highly inter-dependent on others? Do the identified interfaces help to recover a software architecture with high quality? It is desirable to identify interfaces with low inter-dependency and help to recover architectures of high quality.
- **Complexity of the discovered interface behavioral models.** Does the approach discover simple interface behavioral models? It is desirable to identify interface behavioral models that are simple and easy-to-understand.

Based on these quality criteria, we define the following three quality metrics.

5.2.1 Functional Consistency (FC). Generally speaking, interfaces represent functional aspects of a component. An interface is functionally consistent if all functionally related methods are correctly grouped into it. Manually identifying functionally related methods is labour-intensive and highly subjective. Therefore, we use the following observations to objectively quantify the functional relatedness of methods. If methods are called by the same component, we consider these methods as functionally related from the component point of view. Similarly, if methods are called by the same method, we consider these methods as functionally related from the method point of view. The former defines functionally

[1]http://www.processmining.org/
[2]https://svn.win.tue.nl/repos/prom/Packages/SoftwareProcessMining/

related methods in a relatively coarse-grained way while the latter defines more fine-grained functionally related methods. Based on the above observations, we define the following metrics to measure the functional consistency of interfaces.

- PMC quantifies the proportion of methods that are correctly grouped into one interface according to the caller components. It is defined as the proportion of methods of an interface that are called by one of its caller components. We take the average of all caller components for each interface. This metric is borrowed from [19].
- PMM quantifies the proportion of methods that are correctly grouped into one interface according to the caller methods. It is defined as the proportion of methods of an interface that are called by one of its caller methods. We take the average of all caller methods for each interface.
- PCC quantifies the proportion of methods that are wrongly separated across interfaces according to the caller components. It is defined as the maximal proportion of methods that are called by the same component and that are grouped into the same interface. We take the average of all caller components of the interfaces identified for one component.
- PCM quantifies the proportion of methods that are wrongly separated across interfaces according to the caller methods. It is defined as the maximal proportion of methods that are called by the same method and that are grouped into the same interface. We take the average of all caller methods of the interfaces identified for one component.

Let $C \in COM$ be a software component, SD_C be its execution data, and IS_C be its interface set. To define PMC, PMM, PCC, and PCM, we first introduce the following notation. According to Definition 4, assume that we have a candidate interface $\tilde{I}$ called by $n \in \mathscr{U}_N$, for any $n' \in \tilde{I}$ we define $\omega(n', \tilde{I}) = \{n\}$ is the caller method set of n' in $\tilde{I}$, i.e., it is a set of methods where n' in $\tilde{I}$ is called. $\zeta(n', \tilde{I}) = \{C \in \mathscr{U}_{CO} | \exists m \in SD_C : \widehat{c(m)} = n \wedge \widehat{\eta(c(m))} \in C\}$ is the caller component set of n' in $\tilde{I}$, i.e., it is a set of components where n' in $\tilde{I}$ is called.

Assume $I \in IS_C$ is an interface obtained by merging candidate interfaces $\tilde{I}_1$ and $\tilde{I}_2$ according to Definition 5. For any $n \in I$, its caller method set is defined as follows:

$$\omega(n, I) = \begin{cases} \omega(n, \tilde{I}_1) \cup \omega(n, \tilde{I}_2) \; if\, n \in \tilde{I}_1 \cap \tilde{I}_2 \\ \omega(n, \tilde{I}_1) \; if\, n \in \tilde{I}_1 \wedge n \notin \tilde{I}_2 \\ \omega(n, \tilde{I}_2) \; if\, n \in \tilde{I}_2 \wedge n \notin \tilde{I}_1 \end{cases} \tag{1}$$

Similarly, for any $n \in I_C$, its caller component set is defined as follows:

$$\zeta(n, I) = \begin{cases} \zeta(n, \tilde{I}_1) \cup \omega(n, \tilde{I}_2) \; if\, n \in \tilde{I}_1 \cap \tilde{I}_2 \\ \zeta(n, \tilde{I}_1) \; if\, n \in \tilde{I}_1 \wedge n \notin \tilde{I}_2 \\ \zeta(n, \tilde{I}_2) \; if\, n \in \tilde{I}_2 \wedge n \notin \tilde{I}_1 \end{cases} \tag{2}$$

For any $I \in IS_C$, $CM(I) = \bigcup_{n \in I} \omega(n, I)$ is its caller method set and $CC(I) = \bigcup_{n \in I} \zeta(n, I)$ is its caller component set. $CMS = \bigcup_{I \in IS_C} CM(I)$ is the caller method set of component C and $CCS = \bigcup_{I \in IS_C} CC(I)$ is the caller component set of component C.

$$PMC = \frac{\sum_{I \in IS_C} \frac{\sum_{C \in CC(I)} \frac{|M_C(C, I)|}{|I|}}{|CC(I)|}}{|IS_C|} \tag{3}$$

where $M_C(C, I)$ is the set of methods in I that are called by C.

$$PMM = \frac{\sum_{I \in IS_C} \frac{\sum_{n \in CM(I)} \frac{|M_M(n, I)|}{|I|}}{|CM(I)|}}{|IS_C|} \tag{4}$$

where $M_M(n, I)$ is the set of methods in I that are called by n.

$$PCC = \frac{\sum_{C \in CCS} \frac{max\{|M_C(C, I)| | I \in IS_C\}}{|\bigcup_{I \in IS_C} M_C(C, I)|}}{|CCS|} \tag{5}$$

$$PCM = \frac{\sum_{n \in CMS} \frac{max\{|M_M(n, I)| | I \in IS_C\}}{|\bigcup_{I \in IS_C} M_M(n, I)|}}{|CMS|} \tag{6}$$

PMC and PMM measure to what extent functionally related methods are put together in the same interface. A high PMC(PMM) indicates that the interface methods are called by all the interface caller components(methods) and therefore are correctly grouped into one interface. A low PMC(PMM) indicates that the interface methods are functionally unrelated and are wrongly grouped into the same interface. On the other hand, PCC and PCM measure the functional quality of separating methods across different interfaces. A high PCC(PCM) means that a high proportion of methods are correctly separated into multiple interfaces while a low PCC(PCM) indicates that a high proportion of related methods are wrongly separated into multiple interfaces.

To handle the multiplicity of the metrics, we summarize these four metrics as the Functional Consistency (FC) as follows: (1) compute the harmonic mean of PCM and PMM as PM based on Eq. 7; (2) compute the harmonic mean of PMC and PCC as PC based on Eq. 8; (3) compute the harmonic mean of PM and PC as FC based on Eq. 9. Note that PM and PC measure the functional consistency of interfaces from both component and method perspectives.

$$PM = \frac{2 \times PCM \times PMM}{PCM + PMM} \tag{7}$$

$$PC = \frac{2 \times PMC \times PCC}{PMC + PCC} \tag{8}$$

$$FC = \frac{2 \times PC \times PM}{PC + PM} \tag{9}$$

5.2.2 *Functional Inter-dependency (FI).* The functional interdependency metric indicates how tightly one interface interacts with the others. It is an indicator of the quality of the recovered software architecture. A low FI value means that the quality of the architecture is high. The FI of an interface is defined as the ratio of the number of interfaces it interacts with to the total number of interfaces (ignoring itself) in the software system. Given an interface I, its FI metric can be computed as follows:

$$FI = \frac{Interacting(I)}{Sum - 1} \tag{10}$$

Table 4: Comparison of Interface Identification Approaches

Approach	Interface Identification Strategy
A1	interfaces are identified by grouping all methods of the component.
A2 [1]	interfaces are identified by grouping methods of the same class.
A3 [19]	interfaces are identified by grouping methods that are called by the same component.
A4	interfaces are identified by grouping methods that are called by the same method of another component.

where *Interacting*(I) represents the number of interfaces that interacts with I and *Sum* is the total number of identified interfaces in the software system.

5.2.3 *Complexity.*

The complexity of the identified interface behavioral models plays an important role in the comprehension and maintenance of the software architecture. A simple model allows a quick understanding of the interface behavior, and therefore a better and time-efficient maintenance. To quantify the complexity of the discovered interface behavioral models, we use the following two complexity metrics with the assumption that the interface behavioral models are represented by workflow nets [5].

- **Extended Cardoso metric (ECaM).** The ECaM metric measures the structural complexity of the model by counting the various splits (XOR,OR and AND) in the net and gives each of them a certain penalty.
- **Extended Cyclomatic metric (ECyM).** The ECaM metric measures the behavioral complexity of the model by analyzing its reachability graph.

The complexity metric is defined as the harmonic mean of ECaM and ECyM metrics. A low complexity value indicates that the interface behavior is simple.

$$Complexity = \frac{2 \times ECaM \times ECyM}{ECaM + ECyM} \tag{11}$$

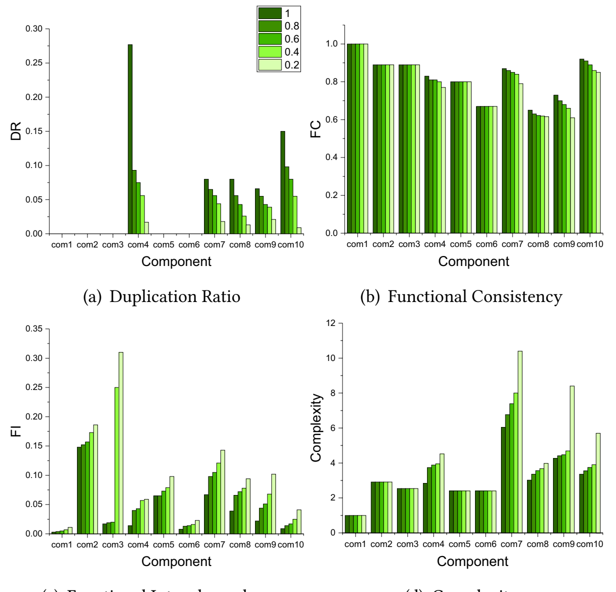
In the following, the *FC*, *FI* and *Complexity* metrics are provided for each component by taking their average for all the interfaces.

5.3 Existing Interface Identification Approaches

We compare our approach with three interface discovery approaches. The first approach (denoted as **A1**) is a naive baseline that simply suggests a single interface that includes all methods of the component. The second approach (denoted as **A2**) is proposed by *Simon et al.* [1] and is structure-oriented. This approach identifies interfaces by grouping methods that belong to the same class, i.e., one interface is created for each class if the class has methods that are used by other components. The third one (denoted as **A3**) is presented by *Seriai et al.* [19] and is function-oriented. It identifies the interfaces of a component by grouping methods that are called by the same component, i.e., each set of methods called by the same components forms one interface. Our approach that is also function-oriented is denoted as **A4**. The main differences of these four interface identification approaches are summarized in Table 4.

5.4 Similarity Threshold Evaluation for A4

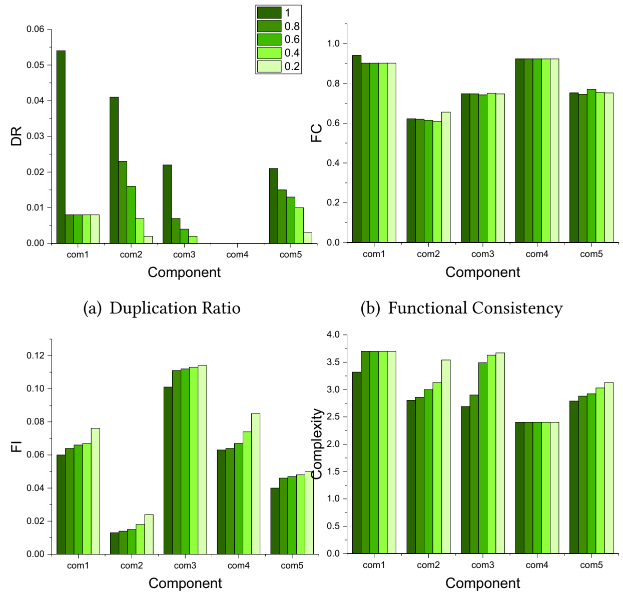
In this first experiment, we study how the similarity threshold affects the quality of our discovered interfaces. As discussed in



(a) Duplication Ratio (b) Functional Consistency

(c) Functional Inter-dependency (d) Complexity

Figure 4: Different Similarity Thresholds Comparison of A4 Using JGraphx



(a) Duplication Ratio (b) Functional Consistency

(c) Functional Inter-dependency (d) Complexity

Figure 5: Different Similarity Thresholds Comparison of A4 Using JHotDraw

Section 3.2, the similarity between interfaces is measured by the *Jaccard distance* of their method sets and only interfaces whose similarity is above a certain threshold can be merged. The interface merging allows reducing the method duplication ratio among interfaces of the same component.

Given a component C and its interface set IS_C, the duplication ratio among interfaces of C is computed as follows:

$$DR = \frac{\sum\limits_{1 \leq i < j \leq |IS_C|} Sim(I_i, I_j)}{|IS_C| \times (|IS_C| - 1)} \tag{12}$$

where $Sim(I_i, I_j)$ represents the similarity between I_i and I_j as defined in Definition 5.

For the three open-source software systems, we first construct their component configurations by using the package information, i.e., classes of the same package are grouped as a component. Then we identify interfaces for each component of *GoogleSheet*, *JGraphx* and *JHotDraw* using **A4** with similarity threshold values 0.2, 0.4, 0.6, 0.8, and 1.0. Afterwards, the quality of interfaces are measured using the quality metrics introduced in Section 5.2. By exploring the interfaces identified from *GoogleSheet*, we found that there is no duplication among them. Therefore, there is no need to measure the effect of similarity on the interface identified. The evaluation results of *JGraphx* and *JHotDraw* are shown in Figs. 4-5.

Figs. 4-5 (a) show the results of duplication ratio for *JGraphx* and *JHotDraw*. We see that the duplication ratio per component decreases as the similarity threshold value decreases. This is because with the decrease of similarity threshold, more and more duplicated interfaces can be merged. Exceptionally, we can see that there is no duplication for components *com*1, *com*2, *com*3, *com*5, and *com*6 of *JGraphx* and component *com*4 of *JHotDraw*. Therefore, their FC, FI and complexity values of different similarity thresholds keep unchanged. On the other hand, the FC values in Figs. 4-5 (b) show that, in general, the functional consistency decreases (or the proportion of functionally unrelated methods per interface increases) as the similarity threshold value decreases. As for the FI and complexity metrics, they increase as the similarity threshold value decreases as shown in Figs. 4-5 (c)-(d). The rationale behind is that the more interfaces are merged, the more functionally unrelated methods each interface contains, and therefore the FI and complexity values per interface also increase.

Different from the general conclusion that the FC decreases as the similarity threshold value decreases, there are some exceptions. Considering for example component *com*2 of *JHotDraw* as shown in Fig. 5 (b). There is an increase of the FC value from similarity threshold 0.4 to similarity threshold 0.2. To explain the reason of such type of exception, we need to understand the meaning of FC. It is a compound metric of PMC, PMM, PCC, and PCM according to Eqs.7-9. We have closely inspected the values of PMC, PMM, PCC, and PCM for each component of *JHotDraw* (due to space limitations, we cannot include all the detailed values in the paper). The exceptional behavior of FC in Fig. 5 (b) is caused by the abnormal change of PMC. There is a sudden increase of the PMC value from similarity threshold 0.4 to similarity threshold 0.2 for *com*2. The reason for this sudden increase is that the merged interfaces have the same caller component, thus the proportion of functionally related methods from component point of view does not decrease as expected. This also explains the exceptions of *com*3 from similarity threshold 0.4 to similarity threshold 0.2 and *com*5 from similarity threshold 0.6 to similarity threshold 0.4.

In summary, there is a clear trade-off between reducing duplication among interface, and ensuring a high quality of identified interfaces. Event though similarity threshold 1 can maintain the best quality of identified interfaces, but it leads to the highest duplication ratio compared with other similarity thresholds. The use of similarity threshold makes our approach more flexible and applicable in real-life scenarios. It helps users to find a balance between the duplication among interfaces, as well as the functional consistency, functional inter-dependency and complexity of discovered interfaces.

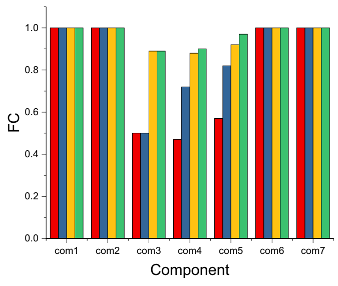
5.5 Comparative Evaluation and Discussion

In this section, we compare the quality of the interfaces identified by different approaches. More specifically, we first identify interfaces for each component of the three software systems using approaches **A1-A4**. For **A4**, we set the similarity threshold value to 0.8 because this value provides a good trade-off between the quality and duplication according to our previous discussion. Afterwards, the quality of identified interfaces is measured using the metrics defined in Section 5.2. Detailed evaluation results are shown in Figs. 6-7.

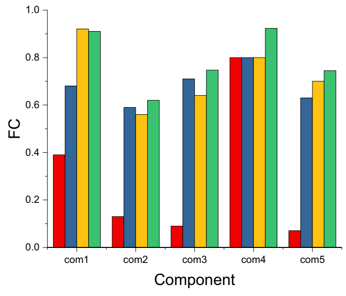
Figs. 6 shows the evaluation results in terms of functional consistency (FC) for *GoogleSheet*, *JGraphx* and *JHotDraw*. The FC values of **A1**-**A2** are much lower than those of **A3**-**A4**. Exceptionally, we can see that the FC values of **A1**-**A4** are the same for components *com*1, *com*2, *com*6 and *com*7 of *GoogleSheet* and component *com*1 of *JGraphx*. This is because these components only have one method. The lower values observed by **A1**-**A2** can be explained by the fact that these two approaches define interfaces based on the internal structure of components (i.e., belonging component/class) rather than based on their functional aspects. Compared with **A1**-**A2**, **A3**-**A4** can guarantee relatively high FC values as these two approaches are more function-oriented. More specifically, **A3** identifies interfaces by clustering functionally related methods from component point of view while **A4** identifies interfaces by clustering functionally related methods from method point of view. Therefore, **A3** can guarantee a perfect PC (i.e., all methods of the same interface are functionally consistent from the component point of view) and **A4** can guarantee a perfect PM (i.e., all methods of the same interface are functionally consistent from the method point of view). As for the FC values, **A4** slightly outperforms **A3** as shown in Fig. 6. This can be explained by the fact that some methods that are considered functionally related from the component point of view are not really related if we look from the method point of view.

Figs. 7 shows the functional inter-dependency (FI) metric results for three software systems. A high FI value means high coupling among the identified interfaces which indicates low architecture quality. This metric measures the quality of the recovered architecture based on different interface identification strategies. Generally speaking, the FI values of **A1**-**A2** are much higher than those of **A3**-**A4**. This observation is in line with the observation we made earlier for FC values. Therefore, a high/low FC is usually correlated with a low/high FI. As for **A3** and **A4**, **A3** always lead to higher FI values than that of **A4**. The rationale behind is that the functionality of the identified interfaces based on **A3** are more general and typically coupling with more interfaces than **A4**.

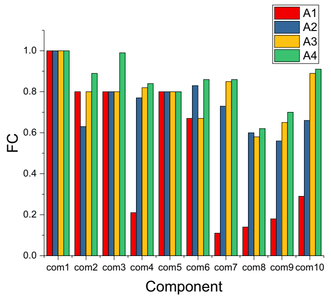
Figs. 8 shows the complexity evaluation results for *GoogleSheet*, *JGraphx* and *JHotDraw*. This metric measures the complexity of the discovered interface behavioral models. A low complexity value



(a) GoogleSheet FC

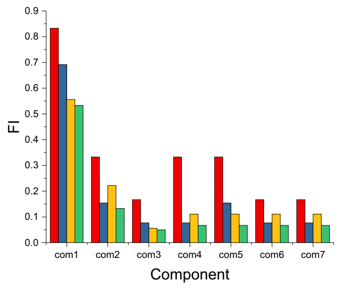


(b) JHotDraw FC

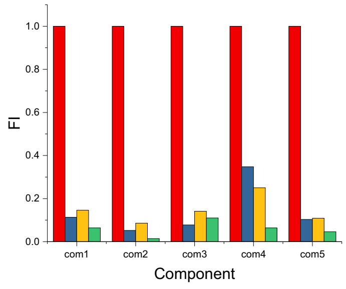


(c) JGraphx FC

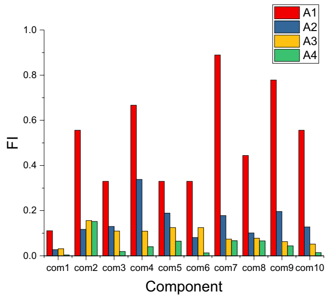
Figure 6: Comparison of FC Metric for Three Software



(a) GoogleSheet FI

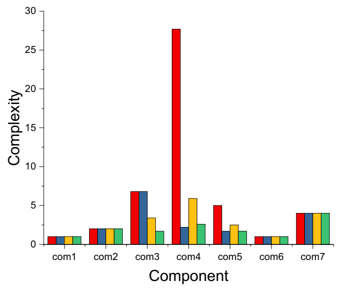


(b) JHotDraw FI

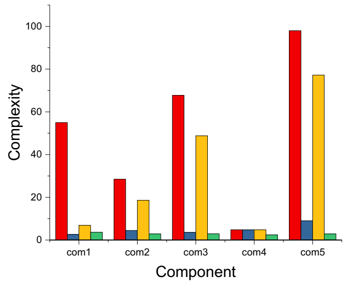


(c) JGraphx FI

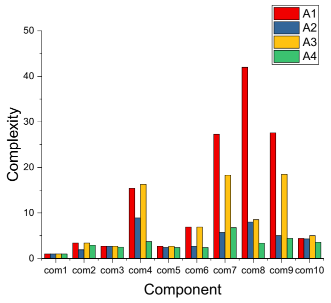
Figure 7: Comparison of FI Metric for Three Software



(a) GoogleSheet Complexity



(b) JHotDraw Complexity



(c) JGraphx Complexity

Figure 8: Comparison of Complexity Metric for Three Software

Table 5: Comparison of Different Approaches

	A1	A2	A3	A4
FC	low	low	relatively high	high
FI	high	high	relatively low	low
Complexity	high	low	high	low

means that the interface has simple behavior. Generally speaking, the complexity values of **A1** and **A3** are much higher than those of **A2** and **A4**. This is because **A2** and **A4** define interfaces based on some fine-grained heuristics (i.e., caller method and belonging class) while the heuristics used for **A1** and **A3** are much coarse-grained (i.e., caller/belonging component). As for **A2** and **A4**, **A2** leads to more complex interfaces than those of **A4** for most of the cases according to Fig. 8 because method-level heuristics are often more fine-grained than class-level ones.

Table 5 summarizes the comparison results of these four approaches in terms of FC, FI, and complexity. In summary, compared with existing interface identification approaches **A1**-**A3**, our approach can (1) discover interfaces that are more functionally consistent and simple; and (2) help to reconstruct the software architecture with better quality.

5.6 Visual Comparison of Interface Behavioral Models

A1-**A3** only return the method set of each identified interface without demonstrating how this interface actually behaves. After identifying interfaces, we discover a behavioral model to help visualizing its behavior. By taking the execution data and the component configuration as input, users can run our *Software Interface Behavior Discovery* plugin, and then the interface behavioral models of each component are returned. Our tool also supports the discovery of behavior models for interfaces that are identified by other approaches. In this section, we visually compare interfaces identified by **A3** and **A4** which are more function-oriented.

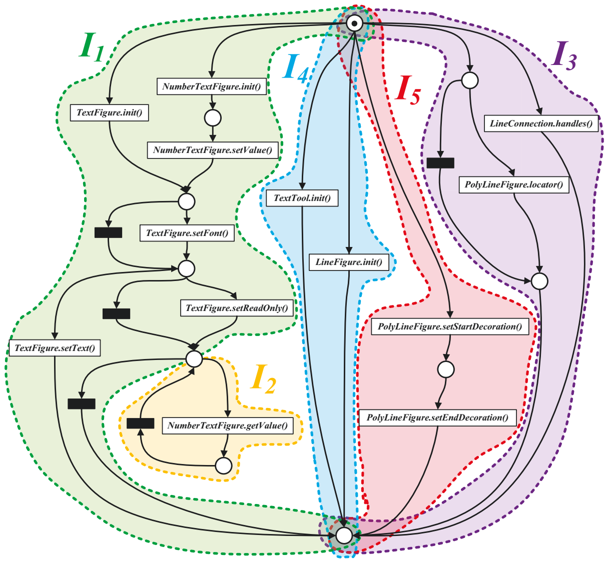


Figure 9: Interface Behavioral Model of I_0 in *Com1* from *JHotDraw* using *A3*

Fig. 9 shows the behavioral model of an interface (denoted as I_0) in component *Com1* of *JHotDraw* discovered by **A3**. The discovered models are redrawn to improve readability. I_0 encapsulates a number of drawing functions, e.g., figure decoration, figure resize and figure position location. Fig. 10 shows the behavioral models of five interfaces (denoted as I_1-I_5) in component *Com1* of *JHotDraw* discovered by **A4**. Each interface provides a relatively independent drawing function and all functions provided by I_1-I_5 are fully covered by I_0. The functional mappings from I_0 to I_1-I_5 are indicated in Fig. 9. More specifically, I_1 refers to the highlighted branch using green and it provides text figure setting function. I_2 (highlighted in yellow) is used to get value from a text figure. I_3 (highlighted in purple) is used to resize a figure resize and locate position of a polyline figure. I_4 (highlighted in blue) provides a factory to initialize line figures and text tools. I_5 (highlighted in red) is used to decorate a polyline figure.

Compared with I_1-I_5, I_0 contains many methods that are functionally unrelated (from the method point of view) and provides a

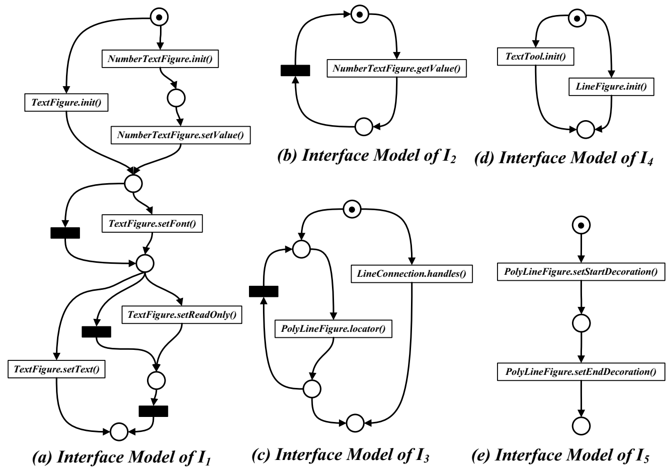

NumberTextFigure.init()
TextFigure.init()
NumberTextFigure.setValue()
TextFigure.setFont()
TextFigure.setReadOnly()
TextFigure.setText()
(a) Interface Model of I_1
NumberTextFigure.getValue()
(b) Interface Model of I_2
LineConnection.handles()
PolyLineFigure.locator()
(c) Interface Model of I_3
TextTool.init()
LineFigure.init()
(d) Interface Model of I_4
PolyLineFigure.setStartDecoration()
PolyLineFigure.setEndDecoration()
(e) Interface Model of I_5

Figure 10: Interface Behavioral Models of the *Com1* from *JHotDraw* using *A4*

number of functions. This further supports our conclusion in the previous section stating that **A4** leads to interfaces with higher functional consistency and lower complexity compared with **A3**.

5.7 Threads to Validity

In the following, we discuss the main threats that may affect the validity of our approach.

- Similar to other dynamic analysis techniques, the quality of the proposed approach heavily depends on the completeness of the software execution data. Therefore, the suitability of the selected monitoring scenarios directly influence the quality of the analysis results [17].
- An implicit assumption of the experimental evaluation is that classes in a package are grouped in a component. The results may be biased if this assumption does not hold.

6 CONCLUSION

By exploiting the tremendous amounts of software execution data, we can discover a set of interfaces and their behavioral models for a given software component. Our approach has been implemented in the open source process mining toolkit ProM and its advantages and applicability were demonstrated by applying it to a set of software execution data generated by three different software cases as well as comparing it with three existing interface identification approaches.

This paper provides initial results in the area of inter-component interaction by discovering a set of interfaces for each component. It answers questions like what functions can be provided by a certain component. A future challenge is to discover how components interact with each other during execution in real-life applications. In addition, the scalability is another pressing issue to be solved. We would like to apply our approach to large-scale real-life software systems that may involve millions of method calls to evaluate the stability of the approach.

REFERENCES

[1] Simon Allier, Salah Sadou, Houari Sahraoui, and Régis Fleurquin. 2011. From object-oriented applications to component-oriented applications via component-oriented architecture. In *9th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 214–223.

[2] Kim Soo Dong and Chang Soo Ho. 2004. A systematic method to identify software components. In *11th Asia-Pacific Software Engineering Conference*. IEEE, 538–545.

[3] David Garlan. 2000. Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. ACM, 91–101.

[4] Selim Kebir, Abdelhak-Djamel Seriai, Sylvain Chardigny, and Allaoua Chaoui. 2012. Quality-centric approach for software component identification from object-oriented code. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*. IEEE, 181–190.

[5] Kristian Bisgaard Lassen and Wil MP van der Aalst. 2009. Complexity metrics for Workflow nets. *Information and Software Technology* 51, 3 (2009), 610–626.

[6] Maikel Leemans and Cong Liu. 2017. XES Software Event Extension. *XES Working Group* (2017), 1–11.

[7] Maikel Leemans and Wil van der Aalst. 2015. Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In *18th International Conference on Model Driven Engineering Languages and Systems*. IEEE, 44–53.

[8] Sander JJ Leemans, Dirk Fahland, and Wil van der Aalst. 2013. Discovering block-structured process models from event logs-a constructive approach. In *Application and Theory of Petri Nets and Concurrency*. Springer, 311–329.

[9] Cong Liu, Hua Duan, and et al. Qingtian, Zent. 2016. Towards Comprehensive Support for Privacy Preservation Cross-organization Business Process Mining. *IEEE Transactions on Services Computing* (2016).

[10] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil van der Aalst. 2016. Component Behavior Discovery from Software Execution Data. In *International Conference on Computational Intelligence and Data Mining*. IEEE, 1–8.

[11] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil van der Aalst. 2018. A Framework to Support Behavioral Design Pattern Detection from Software Execution Data. In *13th International Conference on Evaluation of Novel Approaches to Software Engineering*. 1–12.

[12] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil van der Aalst. 2018. A General Framework to Detect Behavioral Design Patterns. In *International Conference on Software Engineering*. ACM, 1–2.

[13] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil van der Aalst. 2018. Software Architectural Model Discovery from Execution Data. In *13th International Conference on Evaluation of Novel Approaches to Software Engineering*. 1–8.

[14] GuanJun Liu. 2014. Some complexity results for the soundness problem of workflow nets. *IEEE Transactions on Services Computing* 7, 2 (2014), 322–328.

[15] SK Mishra, Dharmender Singh Kushwaha, and Arun Kumar Misra. 2009. Creating Reusable Software Component from Object-Oriented Legacy System through Reverse Engineering. *Journal of object technology* 8, 5 (2009), 133–152.

[16] Hiroshi Nagamochi and Toshihide Ibaraki. 2008. *Algorithmic aspects of graph connectivity*. Vol. 123. Cambridge University Press New York.

[17] Ana Nicolaescu, Horst Lichter, and Veit Hoffmann. 2017. On Adequate Behavior-based Architecture Conformance Checks. In *24th Asia-Pacific Software Engineering Conference (APSEC 2017)*. IEEE, 1–10.

[18] Vladimir Rubin, Christian Günther, Wil van der Aalst, Ekkart Kindler, Boudewijn van Dongen, and Wilhelm Schäfer. 2007. Process mining framework for software processes. In *Software Process Dynamics and Agility*. Springer, 169–181.

[19] Abderrahmane Seriai, Salah Sadou, Houari Sahraoui, and Salma Hamza. 2014. Deriving component interfaces after a restructuring of a legacy system. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*. IEEE, 31–40.

[20] Anas Shatnawi, Abdelhak-Djamel Seriai, Houari Sahraoui, and Zakarea Alshara. 2017. Reverse engineering reusable software components from object-oriented APIs. *Journal of Systems and Software* 131 (2017), 442–460.

[21] Wei Song, Hans-Arno Jacobsen, Chunyang Ye, and Xiaoxing Ma. 2016. Process discovery from dependence-complete event logs. *IEEE Transactions on Services Computing* 9, 5 (2016), 714–727.

[22] Chintakindi Srinivas, Vangipuram Radhakrishna, and CV Guru Rao. 2014. Clustering and classification of software component for efficient component retrieval and building component reuse libraries. *Procedia Computer Science* 31 (2014), 1044–1050.

[23] Ducasse Stéphane and Pollet Damien. 2009. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering* 35, 4 (2009), 573–591.

[24] Wil van der Aalst. 2016. *Process Mining: Data Science in Action*. Springer.

[25] Wil van der Aalst, Anna Kalenkova, Vladimir Rubin, and Eric Verbeek. 2015. Process discovery using localized events. In *International Conference on Applications and Theory of Petri Nets and Concurrency*. Springer, 287–308.

[26] Wil MP Van der Aalst. 1998. The application of Petri nets to workflow management. *Journal of circuits, systems, and computers* 8, 01 (1998), 21–66.

[27] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and Wil van der Aalst. 2005. The ProM Framework: A New Era in Process Mining Tool Support. In *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets (ICATPN'05)*. Springer-Verlag, Berlin, Heidelberg, 444–454.

[28] Hironori Washizaki and Yoshiaki Fukazawa. 2005. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer programming* 56, 1-2 (2005), 99–116.

[29] QingTian Zeng, FaMing Lu, Cong Liu, Hua Duan, and ChangHong Zhou. 2015. Modeling and verification for cross-department collaborative business processes using extended petri nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 45, 2 (2015), 349–362.

[30] Qingtian Zeng, Sherry X Sun, Hua Duan, Cong Liu, and Huaiqing Wang. 2013. Cross-organizational collaborative workflow mining from a multi-source log. *Decision Support Systems* 54, 3 (2013), 1280–1301.