Feature-level Phase Detection for Execution Trace Using Object Cache

Yui Watanabe Takashi Ishio Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
{wtnb-y, ishio, inoue} @ist.osaka-u.ac.jp

ABSTRACT

Visualizing collaborations of objects is important for developers testing and debugging an object-oriented program. Many techniques and tools are proposed to visualize dynamic collaborations involved in an execution trace of a system, however, some execution trace is too large to be transeformed into a single diagram. In this paper, we propose a novel approach to efficiently detecting phases, or high-level behavioral units of interest to developers, using a LRU cache for observing a working set of objects. Our idea is based on the nature of object-oriented programming; a phase starts with preparing objects for the phase and ends with destroying unnecessary objects. Our technique uses a LRU cache for objects to detect a phase transition if the cache is frequently updated. We have applied our approach to two industrial applications and found that our approach detects feature-level phases without the deep knowledge on target applications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Tracing*

General Terms

Algorithm, Experimentation

Keywords

phase detection, dynamic analysis, execution trace, Java program

1. INTRODUCTION

Visualizing collaborations of objects is important for developers testing and debugging an object-oriented program. This is because understanding the behavior of an object-oriented system is more difficult than understanding its structure [1, 27], and because collaborations of objects provide a larger unit of understanding than classes [16]. Many techniques and tools are proposed to visualize dynamic collaborations involved in an execution trace [1, 5, 7, 13, 22, 24].

An important issue in this research area is how to handle a huge amount of events included in an execution trace. One approach is summarizing an execution trace [7, 14]. Another is visualizing an overview of a trace using zoom-in/out functionality [13] or a new viewer named Circular Bundle View [4]. Shimba [22] and JIVE [5] provides query-based interface for visualizing events of interest to developers.

We propose a *phase detection* approach to dividing a long execution trace into several phases before such summarization and visualization. A *phase* relates to what the program is doing at a high level, e.g. reading input, processing a command, accessing a database, waiting for a connection, or computing some set of values [15]. Our method identifies a phase as a consecutive sequence of runtime events. Some phase corresponds to a feature, which is a realized functional requirement of a system [6, 18]. Some other phase may represent a minor phase, or one of the tasks to achieve a feature.

Detecting phases from an execution trace helps developers focus on a small portion of the execution trace. For example, a bug report such as "this program crashed during the login phase" indicates a good clue for developers. Cornelissen reported that filtering out set-up and tear-down phases could improve the readability of a sequence diagram generated from an execution trace of a unit test [3]. Cornelissen's work is based on the naming convention and the behavioral patterns of JUnit testing framework, therefore, it is hard to apply other general applications.

We propose a novel approach to detecting phases involved in execution traces. Our technique is based on the nature of object-oriented programs; many objects are created to achieve a task and most of the objects are destroyed after the task [10, 25]. We employ a LRU cache for observing objects that are working for the current phase; if the cache is frequently updated, we recognize that a new phase is beginning and preparing objects for the new phase. Our goal is a kind of feature location [6] in an execution trace. This is different from phase detection techniques in code optimization and performance analysis area [9, 12, 15].

Our approach does not require the deep knowledge on a target system. As a case study, we have analyzed several use

case scenarios on an industrial system. We found that if we divided an execution trace into 10 phases, 8 of 10 detected phases are correct on average; they covers 93% of feature-level phases and 48% of minor phases in features. The detected phases are good clues for developers to investigate the execution trace. We have also analyzed five programs implementing the same specification developed in a training program of software development. We found that the phases detected from an implementation are similar to the phases detected from another implementation. The result shows our approach is insensitive to the implementation detail of a system.

We integrated the approach into Amida, our sequence diagram visualization tool [24]. Amida automatically detects phases in an execution trace and visualizes each phase as a sequence diagram. Amida also implements several rules to detect loops and recursive calls in execution traces so that a phase is visualized as a compact diagram.

The contributions of this paper are following:

- We propose a lightweight phase detection approach using a cache algorithm. This approach divides an execution trace into a sequence of phases. Our detection approach is based on the nature of object-oriented programming; a phase creating and destroying a large number of objects. Each of output phases corresponds to a feature or one of tasks to achieve a feature.

- We have applied our approach to an industrial application. We have recorded execution traces for four use-case scenarios of the system. We found that if we divided an execution trace into 10 phases, 8 of 10 detected phases are correct on average; they covers 93% of feature-level phases and 48% of minor phases described in use-case scenarios. The detected phases are good clues for developers to investigate the execution trace.

- We also applied our approach to five programs implementing the same specification developed in a training course of software development. We have executed the one use-case on them and compared the resultant phases in the execution traces. We found that the phases detected from an implementation are similar to the phases detected from another implementation. The result shows our approach is insensitive to the implementation detail of a system.

- We have implemented our analysis for Java programs using JVMTI. The approach is integrated into Amida, our sequence diagram visualization tool. Amida enables developers to visualize the behavior of a feature.

The rest of this paper describes the detail of our phase detection approach. Section 2 describes the definition of phase. Our phase detection algorithm is presented in Section 3. Section 4 shows two case studies on industrial applications. In Section 5, we describe the conclusion and future direction of the research.

2. BACKGROUND

Visualization of Dynamic Behavior. Visualizing collaborations of objects is important for developers testing and debugging an object-oriented program. In general, object-oriented programs are difficult to maintain because of dynamic binding of method calls [27]. While collaborations of objects provide a larger unit of understanding than classes [16], reverse engineering and understanding the behavior of an object-oriented system is more difficult than understanding its structure [1].

To support understanding the dynamic behavior of a program, many tools are proposed to visualize dynamic collaborations in a program [1, 5, 7, 13, 22, 24]. An important issue in this research area is how to handle a huge amount of events included in an execution trace. We categorize related work into three approaches.

One approach is summarizing a whole execution trace. Hamou-Lhadj proposed a *utilityhood* function to filtering out utility-like method calls that are less important in general [7]. Reiss proposes to compress an execution trace into a compact representation [14]. Several visualization tools visualize repeated method calls in a compact style [1, 13, 24].

Another approach is visualizing an overview of a trace. Pauw uses zoom-in/out functionality [13]. Cornelissen proposes a new viewer named Circular Bundle View [4]. These views allow developers to investigate a trace in a top-down style.

The third approach is visualizing only method calls of interest to developers. DRT, a design recovery tool, supports automatic selection of method calls related to a user action in graphical user interface [2]. Shimba [22] and JIVE [5] provides query-based interface for visualizing events of interest to developers. Sharp proposes an interactive exploration of UML sequence diagram using both zooming and various filtering facilities [19]. Briand developed a tool to visualize only method calls related to Remote Method Invocation in a distributed system [1].

We propose a novel *phase detection* technique to dividing a long execution trace into several phases before such summarization and visualization described above. Our technique is involved in the third approach; it enables developers to visualize and investigate only a small portion of an execution trace of interest to the developers. Our technique can collaborate with the previous approaches for visualization, for example, we have integrated the technique to Amida, our sequence diagram visualization tool [24].

Phase Detection. Our phase is defined as a consecutive sequence of runtime events. Some phase corresponds to a feature, which is a realized functional requirement of a system [6, 18]. Some other phase may represent a minor phase, or one of the tasks to achieve a feature.

In this paper, a *feature-level phase* denotes a phase corresponding to an execution of a feature. A *minor phase* denotes a phase that is one of the tasks to achieve a feature. When we execute a program according to a use case sce-

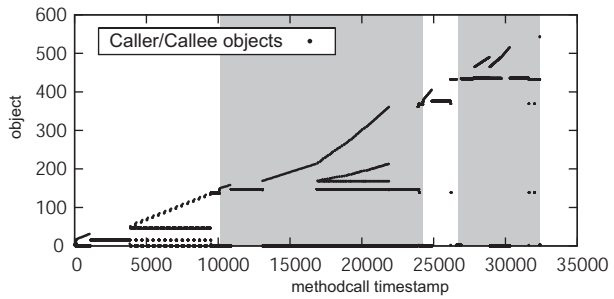


Figure 1: Caller/Callee Objects in a use-case scenario of an industrial system. The execution trace comprises five feature-level phases: login, three steps to update a database record and logout.

nario that is a sequence of features, we get an execution trace including the corresponding feature-level phases. Each feature-level phase involves a sequence of minor phases. Although a minor phase may involve its sub-phases, our main goal is to extract feature-level phases and minor phases described in use-case scenarios.

The phased behavior of a system is easily visualized with a zoom-out view [13]. Figure 1 is an example trace of the phased behavior in an industrial system we have used in the case study. The x-axis of the figure represents a sequence of events. The y-axis plots object IDs of method caller and callee for each method call events. This figure clearly shows that different objects work in the different phases. Developers can easily find phases in the view, however, it is difficult to manually divide the trace into phases without the knowledge of the system.

We propose an automatic phase detection technique using a LRU cache for observing working objects. Our approach is based on two basic hypotheses in object-oriented programming:

- Many objects are created to achieve a task and most of them are destroyed after the task [10, 25]. At the beginning of each phase, new objects are created for the new phase and some objects come from the previous phase [11].
- The beginning and the end of a phase correspond to a method call and a method return event, respectively. For example, a login phase may start with `main` method call and end with a return from `processPassword`.

We have chosen LRU cache rather than other algorithm since LRU is effectively capture local objects working in a short period [21].

Detecting phases is important to understand an execution trace. Cornelissen reported that filtering out set-up and tear-down phases could improve the readability of a sequence diagram generated from an execution trace of a unit test [3]. Cornelissen's work is based on the naming convention and the behavioral patterns of JUnit testing framework, therefore, it is hard to apply other general applications.

```
@1 19 void LoginForm(5).<init>(){
@1 }
@1 20 boolean index_jsp(3).
       _jspx_meth_html_text_0(Tag,PageContext){
@1 21 String LoginForm(5).getShimeiNo(){
@1 }
@1 }
@1 22 boolean index_jsp(3).
       _jspx_meth_html_password_0(Tag,PageContext){
@1 23 String LoginForm(5).getPassword(){
   :
```

```
@0 31 boolean java.util.ArrayList(492).add(java.lang.Object) {
```

Thread ID — Timestamp — Method Signature — Object ID

Figure 2: An example trace

Wang have proposed hierarchical dynamic slicing with another definition of phase based on syntactic structure [26]. The syntax-based phase definition provides hierarchical phases for developers to investiagete the detail of a fault using an execution trace. However, the approach does not support to find a phase corresponding to a feature described in a use-case.

Our goal is a kind of feature location [6] while conventional phase detection techniques proposed in program optimization area also detect phases in the trace. These conventional techniques for program optimization typically use a fixed-length interval (e.g. 10 milliseconds [15]) since performance optmization techniques are applied independently of software features. Some optimization technique recognizes a phase transition between two phases, that is an unstable interval and hard to optimize [12]. Such a model is different from our phase detection.

3. AUTOMATIC PHASE DETECTION

We propose a phase detection technique using a LRU cache for recording objects that are working for the current phase; if the cache is frequently updated, we recognize that a new phase is beginning and preparing objects for the new phase.

Our detection method takes as input an execution trace $E = [e_1, \ldots, e_n]$ where e_k corresponds to a method call event. e_k knows its caller object, callee object and the depth of the call stack at the event. The method outputs phases as a list of timestamps $P = [t_1, t_2, ..., t_p]$ where t_k indicates the beginning of each phase. For example, $P = [1, 30, 50]$ for an execution trace E indicates that the trace E contains three phases $[e_1, \ldots, e_{29}], [e_{30}, \ldots, e_{49}], [e_{50}, \ldots, e_n]$.

3.1 Recording Execution Trace

Our detection method takes as input an execution trace that is a sequence of method call events. Each event has the following attributes.

timestamp represents the sequential order of events.

calleeID denotes which object is called.

procedure DetectPhases(
in $E = [e_1, e_2, \cdots e_{last}]$;
in $c, w, m : integer$; $threshold : double$;
out $P : set\ \ of\ \ timestamp$
(1) C = new LRUCache(c);$P \leftarrow \phi$
(2) **for** t in $[1 \ldots last]$
(3) $updated[t] = update(C, e_t.caller, e_t.callee)$
(4) **if** $frequency(t, w) \geq threshold$
(5) $P \leftarrow IdentifyPhaseHead(t, m)$
(6) **end if**
(7) **end for**

function update(**in** C, callerID, calleeID): integer
1) $b1 \leftarrow C.update(callerID)$ – b1, b2 = true if
2) $b2 \leftarrow C.update(calleeID)$ – C did not contain the ID
3) **if** $b1 \vee b2$ **then return** 1 **else return** 0

function IdentifyPhaseHead(**in** t, m) : integer
1) $min = x = \max(t - m + 1, 1)$
2) **while** $x \leq t$
3) **if** $e_{min}.callstack \geq e_x.callstack$ **then** $min = x$
4) $inc(x)$
5) **end while**
6) **return** min

Figure 3: Phase Detection Procedure

callerID denotes an object which calls a method.

threadID indicates a thread in which the event occurs.

callstack indicates the depth of the call stack for the thread.

These attributes can be retrieved from a sequence of method call/return events. Figure 2 shows an example trace in textual format that is a part of the trace shown in Figure 1. The example includes the end of initialization of forms and the beginning of the login process.

For each method call, we record its object ID, method signature and thread ID. All threads share a common timestamp generator in order to serialize all method call events. To extract all necessary information from a Java program, we are using Amida profiler [24], an implementation of Java Virtual Machine Profiler Interface (JVMTI).

3.2 Phase Detection Algorithm

Our phase detection algorithm is defined as the procedure `DetectPhases` in Figure 3. The procedure takes as input an execution trace E, parameters c, w, m and *threshold*. The four parameters affect the granularity of detected phases. An output $P = [t_1, t_2, ..., t_p]$ is a list of timestamps indicating phases in the trace.

The procedure works as follows:

Table 1: A LRU cache in our algorithm (presented in Figure 3) updated by the example trace. The four parameters are: $c = 3, w = 2, m = 2$ and $threshold = 1.0$.

t	cache	callstack	updated	freq	phase
	[1, 2, 4]	0	0	0	
19	[1, 2, 5]	1	1	0.5	
20	[2, 5, 3]	1	1	1.0	$P \leftarrow 20$
21	[2, 3, 5]	2	0	0.5	
22	[2, 5, 3]	1	0	0	
23	[2, 3, 5]	2	0	0	

1. *Observe the working set of objects using a Least-Recently-Used (LRU) cache.* The LRU cache C keeps a set of object IDs. For each method call event, the cache C is updated by `update` function that calls `C.update`($objID$).

   `C.update` checks whether C contains $objID$ or not. If C does not contain $objID$, C adds $objID$ to the contents, removes the least-recently-used object, and returns true. If C contains the object ID, `C.update` updates the timestamp for $objID$ and returns false.

   `update` function returns 1 if at least one of the caller and callee objects is added to C. Otherwise, the function returns 0.

2. *Detect a phase transition.* We defined $frequency$ of the LRU cache as an indicator of a phase transition.

$$frequency(t, w) = \frac{\Sigma_{x=\max(1,t-w+1)}^{t} updated[x]}{w}$$

   If the $frequency(t, w)$ is higher than a given threshold value, the procedure calls `IdentifyPhaseHead` function to identify the beginning of a new phase.

3. *Identify the head event of a phase.* `IdentifyPhaseHead` function goes back to a method call event that is likely to trigger the new phase. The function identifies the event who has the local-minimum depth of the call stack (the latest one if tied) as the beginning of the phase.

Table 1 shows how the example trace in Figure 2 update a LRU cache. The column t indicates timestamp of each events. The column *cache* represents object IDs contained in the LRU cache. The column *updated* shows the return value of the function `update` in the phase detection procedure. The column $freq$ indicates the value of the frequency function described above. The column *phase* shows when the output phase P is updated. In the example trace, the event whose $t = 20$ results in the maximum update frequency; it triggers `IdentifyPhaseHead` function. `IdentifyPhaseHead` function investigate the depth of the call stack in recent 2 events ($t = 19, 20$) according to the parameter $m = 2$, and select $t = 20$ as the head event of the phase.

Parameters. This algorithm has four parameters: *threshold*, cache size c, window size w for calculating frequency and phase search distance m that specifies how many events are investigated as candidates of the beginning of the phase.

Cache size c specifies the size of the LRU cache C used by `DetectPhases`. The minimum value is 1 and the maximum value equals the number of objects in the input execution trace, respectively.

A smaller cache is more sensitive to changes of a working set; it results in fine-grained (short) phases.

Window size w is used on computation of $frequency(t, w)$. The minimum value is 1 and the maximum value is the size of the execution trace (*last* in the procedure `DetectPhases`), respectively.

A smaller window is more sensitive to changes of a working set.

Frequency threshold *threshold* is compared with $frequency(t, w)$ to detect a phase transition. The minimum value is 0 and the maximum value is 1, respectively.

A lower value is more sensitive to changes of a working set.

Phase search distance m specifies how many events prior to a phase transition point are investigated as candidates of the beginning of the phase.

Computational Complexity. The computational complexity is $O(mn)$ where n is the size of an execution trace and m is a parameter of the algorithm, respectively. This algorithm needs memory for a cache whose size is specified by the parameter c and a window whose size is $max(m, w)$ to keep the events. The algorithm is efficient since c, w and m are much smaller than n.

Handling multithreaded programs. This algorithm can handle a multi-threaded trace in two ways. The one is applying the algorithm for each thread. This is natural if developers would like to investigate a particular thread of control. The other one is using a common LRU cache for all threads and regarding the sum of the depth of all call stacks used in the program as the depth of a virtual single call stack. In the case study of this paper, we took the latter approach.

4. EXPERIMENT

To evaluate the effectiveness of our approach, we have compared phases detected by our approach with phases manually identified by developers. We have used four use-case scenarios for an industrial system, and a use-case scenario for five programs implementing the same specification. We have also investigated how two parameters, cache size c and window size w, affect the granularity of detected phases.

4.1 Settings

We have analyzed two Java applications as follows:

- *Tool Management System* is a web application developed by a software idustry. This system uses a background thread for managing server resources and three threads for proceccing HTTP request. Its size is 37,000 lines of code. The system uses JSP pages as user interface; execution traces record all interaction among JSP pages and business logic classes. We recorded four execution traces correspoinding to four use-case scenarios.
- *Book Management System* is a material for a training course of programming used in an industry. This system is a stand alone application. This system is also multi-threaded; a thread manages a database in the background while a user operate the system. There are five programs implementing the same specification. We have executed the same scenario on each program.

Four scenarios for Tool Management System (T-1, T-2, T-3 and T-4) and a scenario for five implementation of Library Management System (L-1 to L-5) are listed in Table 2. The scenarios for Tool Management System involve several common features such as Login.

We have executed the scenario and recorded execution traces from the systems with a JVMTI profiler [24]. We excluded Java SDK library from the traces. It should be noted that the execution of a feature in a trace may be not identical with the execution of the same feature in another trace according to their scenarios and implementation.

After the execution of the scenarios, we have askd developers of the systems to manually identify phases that represent features in these execution traces. We also asked them to divide a feature-level phase into minor phases that correspond to tasks to achieve the feature. Table 3 shows the number of method call events, objects, and manually identified phases involved in the traces. We also show an example of minor phases in an trace; the following is a list of 18 minor phases to achieve 5 features involved in the trace T-1.

1. Login
   (a) The system shows a login prompt.
   (b) The system checks user name and password input by a user (a manager).
   (c) The system retrieves a list of available tools in the company.
   (d) The system shows a main interface.
2. Listing tools
   (a) The system retrieves its management information.
   (b) The system retrieves a list of requests from developers.
   (c) The system retrieves a list of all tools.
   (d) The system shows the lists to the user.
3. Maintenance view of a tool
   (a) The system retrieves the detail of a tool specified by the user.
   (b) The system shows the detail.

Table 2: Use case sscanarios (feature-level phases) to record execution traces.

ID	Scenario
T-1	Login → Listing tools → Maintenance view of a tool → Updating the tool information → Logout
T-2	Login → Listing tools → Maintenance view of a tool → Updating the tool information → Cancelling the edit → Logout
T-3	Login → Listing tools → Maintenance view of a tool → Logout
T-4	Login → Searching tools → Maintenance view of a tool → System Shutdown
L-1,2,3,4,5	Login → Showing mylist → Adding a new book → Registering a new book → Showing booklist → Searching books → Borrowing a book → Showing the detail of a book → Returning a book → Marking a book → Unmarking a book → Showing browwinglist → Logout → Login as a new user → Logout

Table 3: Execution traces and number of phases detected by developer in those traces.

System	ID	#events	#objects	#feature-level phases	#minor phases
Tool Management System	T-1	32416	546	5	18
	T-2	30494	524	6	19
	T-3	26603	438	4	14
	T-4	15909	237	3	10
Library Management System	L-1	3573	261	15	52
	L-2	3371	272	15	51
	L-3	3797	286	15	51
	L-4	3862	300	15	51
	L-5	4506	341	15	64

   (c) The system also shows commands for maintenance.

4. Updating the tool information
   (a) The system retrieves the detail information of the tool.
   (b) The system registers the updated detail information to the database.
   (c) The system shows the updated detail.
   (d) The system shows commands for maintenance again.
5. Logout
   (a) The system records that the user logged out.
   (b) The system shows a logout message to the user.
   (c) The system cleans up all allocated resources.

We have applied our method to detect phases with various parameter settings and compared the detected phases with the manually identified phases. We fixed two parameters: $threshold = 0.1$ and $m = 700$ because these parameters are less effective on the number of detected phases in all software we have tried (including various industrial and open-source software).

The other two parameters, cache size c and window size w, affect the granularity of detected phases. To detect phases from traces of Tool Management system, we varied cache size c from 10 to the number of objects by 10. And we varied window size w from 10 to 200 by 10 since a large window ($w > 200$) did not affect the result. To detect phases from traces of Library Management System, we varied cache size c from 10 to 350 by 10 since the traces involves at most 350 objects. And we varied window size w from 10 to 300 by 10. Therefore, we have computed phases with 1200 settings for each trace of Tool Management System, 1050 settings for each trace of Library Management System. We took less than five minutes to compute all conbination of parameter settings and exectuion traces on a workstation whose CPU is Xeon 3.0 GHz.

We have detected phases for each trace and collected the number and location of output phases. We have compared the detected phases with the manually identified phases. We evaluated the result based on recall and precision calculated as follows:

$$precision_{[\%]} = \frac{|P \cap Manual|}{|P|}, \quad recall_{[\%]} = \frac{|P \cap Manual|}{|Manual|}$$

P is a set of output phases detected by our method, and $Manual$ is a set of phases manually identified by developers, respectively.

4.2 Result

4.2.1 The Number of Output Phases

This section shows how two parameters, cache size c and window size w, affect the granularity of detected phases. Because of the limited space, here we show only the detailed result of the trace T-1.

Figure 4 shows the number of detcted phases in the trace T-1 for each parameter configuration. A smaller cache size leads frequent cache update, therefore, our procedure outputs smaller (shorter) phases. A smaller window size also

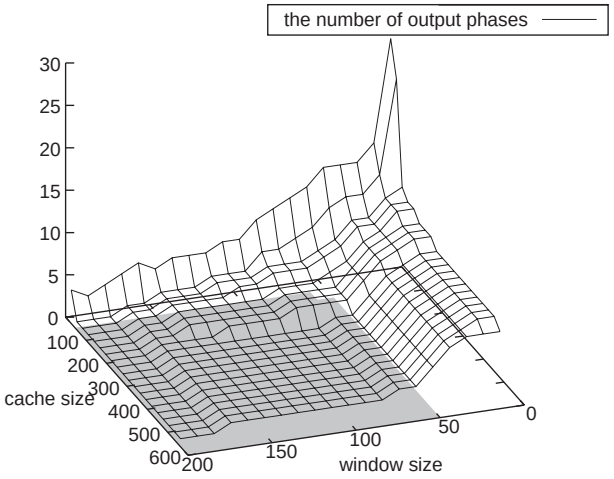


Figure 4: The number of detected phases in the trace T-1 for each parameter setting.

outputs smaller phases since a small window size value regards a small number of new objects as a phase transition.

Figure 5 shows the effect of a single parameter. The top figure shows the result from various cache size c and the fixed window size $w = 50$. A dot plotted at (x, y) indicates that the xth method call in the trace is identified as a phase transition when the cache size is y. Similarly, the bottom figure shows the result from various window size w and the fixed cache size $c = 300$.

A gray bar denotes a feature-level phase transition event that is automatically detected by our method and manually specified by developers. A box denotes a minor phase transition event that is also is detected by both our method and developers. The other dots include false positives of our method and phases that are not recognized by developers.

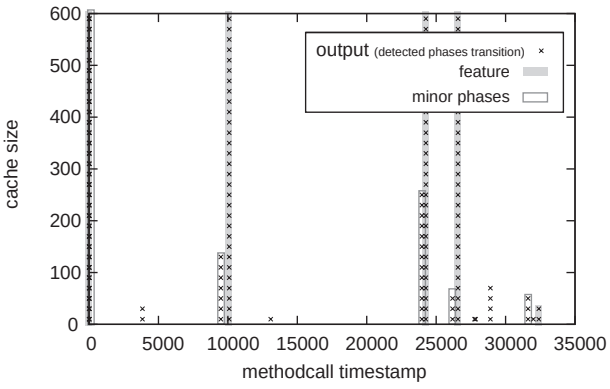
We would like to note that the result of our phase detection is *stable*. If we kept one parameter as a constant value and decreased another parameter, the change always divided a phase to two sub-phases.

In general, a LRU cache is frequently updated at different timestamp if parameters are changed. Nevertheless, our algorithm identifies the same event as the phase head using the depth of a call stack. We found that our approach become unstable when a parameter is extremely small (e.g. cache size $c \leq 20$).

4.2.2 Recall and Precision

This section shows the recall and precision of our approach. According to the limited space, here we show the detailed result of the trace T-1 and the summary for all traces.

Figure 6 shows the average recall and precision for all possible parameter settings that result in the same number of phases in the trace T-1. The x-axis represents the number of phases detected by our approach. "Precision(All)" indicates the average *precision* of all parameter settings that result in the same number of output phases. Our approach



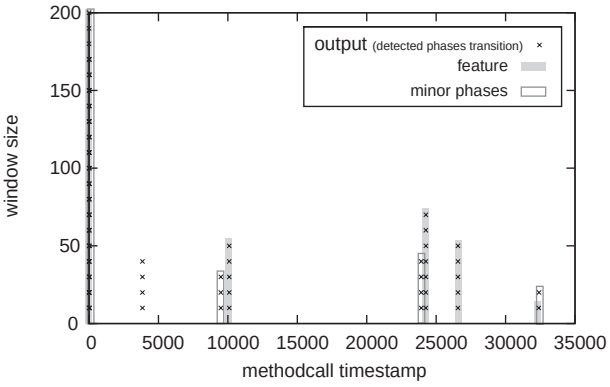


Figure 5: Detected phases in the trace T-1. Window size is fixed ($w = 50$) in the top figure, cache size is fixed ($c = 300$) in the bottom figure.

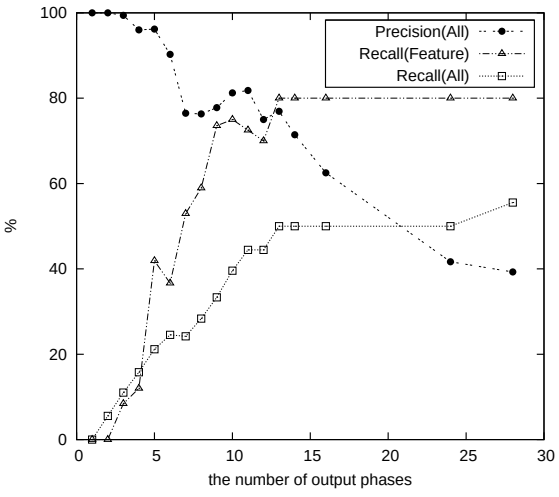


Figure 6: Average recall and precision for various parameter configuration that result in the same number of phases in the trace T-1.

Table 4: Average recall and precision for various parameter configuration that detects the same number of phases.

Tool Management System

#phases	Recall(Feature)	Recall(All)	Precision
5	0.56	0.39	0.93
10	0.90	0.48	0.80

Library Management System

#phases	Recall(Feature)	Recall(All)	Precision
10	0.24	0.20	0.99
15	0.53	0.29	0.98
20	0.45	0.38	0.96

shows high precision when the number of output phases are smaller. Parameter settings that result in precise phases (precision 100%) are indicated by a gray region in Figure 4; the number of output phases is equal to or less than 8 and a parameter is not extremely small.

"Recall(Feature)" indicates how much feature-level phases are detected. "Recall(All)" is calculated for minor phases. Both increase with the number of output phases. However, recall didn't reach 100%: In this case, the maximum "Recall(Feature)" is 80%. Our method never detected the Logout phase with any parameter settings. This is because the Logout phase comprises an extremely small number of objects and method call events. In other case, we also found that our approach is hard to detect a minor phase that is the head of a feature-level phase if the minor phase is extremely small, e.g. an initalization phase of the feature.

We summarized the result for all traces based on recall and precision. Table 4 shows the average recall and precision for all possible parameter configuration except for extremely small parameters, that results in 5 or 10 phases from four traces of Tool Management System, and result in 10, 15 or 20 phases from five traces of Library Management System.

If we got 5 phases with some parameters (arbitrary pair of cache and window size) in one of the four execution traces of Tool Management System, 93% of them are meaningful for developers (7% are false positives); they covers 56% of the features and 39% of the minor phases. 10 phases involve 8 correct phases and 2 false positives on average. This result shows that developers can apply our phase detection approach without the knowledge on a target system and parameter configuration. Although our approach does not cover all of the feature-level and minor phases, the high precision is important for developers to use the result to investigate the execution trace. Our approach detects a phase that is not a feature-level phase, but such a phase is likely to a minor phase that is still a meaningful unit for developers.

4.2.3 *Comparing Different Scenarios*

Here we compare the phases detected with the same parameter setting, cache size $c = 100$ and window size $w = 50$, in two traces T-2 and T-3 from Tool Management System.

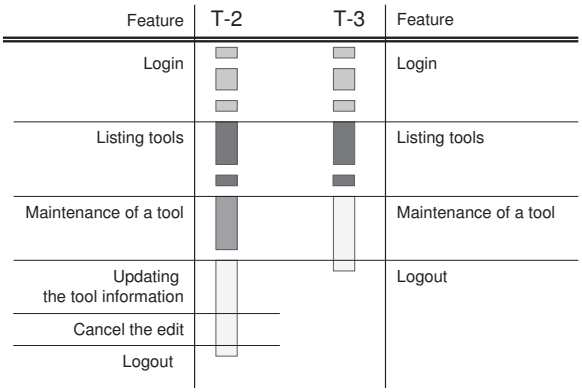


Figure 7: Phases of two traces from Tool Management Systems with parameters $c = 100$ and $w = 50$. A rectangle represents a detected phase, a horizonal line represents a manually identified phase.

Figure 7 shows 7 phases in T-2 and 6 phases in T-3 we have detected. In this case, we have no false positives; all detected phases correspond to minor phases. A rectangle represents a detected phase, a horizonal line represents a manually indentified phase. For example, our method detected 3 phases in the first feature-level phase "Login" ; the first one corresponds one minor pahse "The system shows a login prompt", the second one corresponds two minor phases "The system checks user name and password input by a use" and "The system retrieves a list of available tools in the company", and the third one corresponds one minor phase "The system shows a main interface".

The two traces execute the same features but use different methods and objects in the feature-level phases. Our approach detected the same features involved in the different use-case scenarios.

4.2.4 *Comparing Different Implementation*

We have compared the phases detected with the same parameter setting in five traces of the five implementation of Library Management System. Figure 8 shows the phases detected with the paramters: cache size $c = 150$ and window size $w = 150$. Horizontal lines indicates the beginning of feature-level phases manually identified by developers. Each feature-level phase comprises 2 to 6 minor phases. A vertical bar is one phase detected by our method. There are no false positives in this case.

It should be noted that the internal structure of these programs are different from each other, nevertheless, our technique detected the similar phases. This result shows that our approach is insensitive to the implementation detail of a system. This stability is important property for developers who modify a program (e.g. in debugging process) since it allows developers to compare traces before and after the modification.

4.3 Discussion

Threats to validitiy. Our case study reflects the industrial environment. We have used the industrial systems and use-

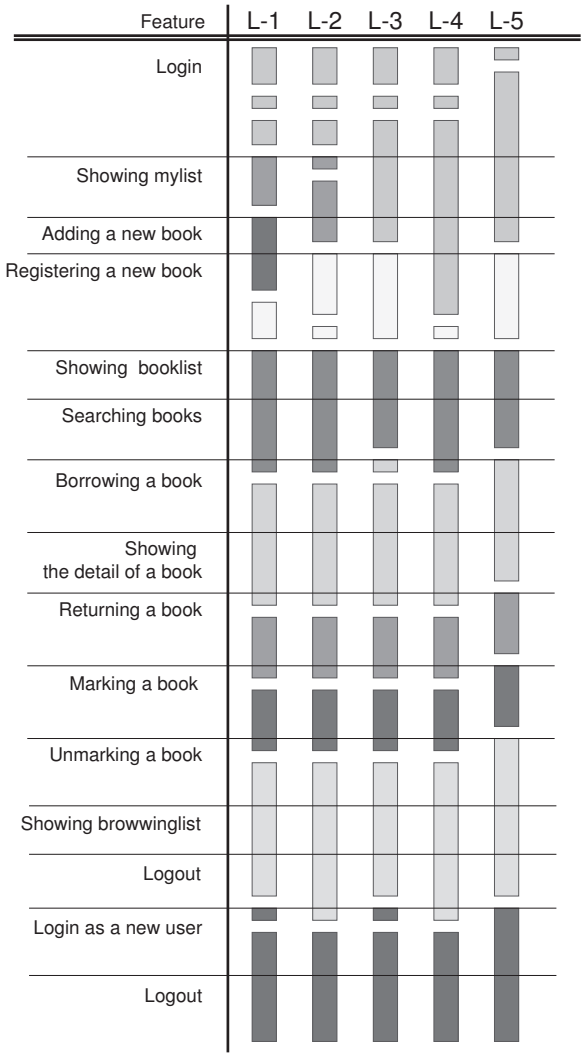


Figure 8: Detected phases in five traces of Library Management Systems with parameters $c = 150$ and $w = 150$.

case scenarios written by the developers of the systems. However, the target domain is limited to enterprise application that interacts with databases. Both systems are implemented as a multi-threaded program, however, the use-case scenarios include no descriptions about the concurrent user-interaction. Therefore, we need further investigation of multi-threaded programs with concurrent interaction scenarios and other programs in different domains.

We have recorded execution traces excluding Java standard library since phases are characterized by application-specific objects rather than generic objects. This filtering may affect the result, however, recording all objects is impractical according to its runtime overhead.

Mapping features to phases. Our approach outputs only a sequence of phases as a list of timestamps. Developers have to manually map features in a use-case scenario to phases in its execution trace. This is not so difficult but tedious since the one scenario may generate various execution traces because of different input and environment variable, for example.

Therefore, we need a way to automatically assign appropriate names to phases. The problem is related to feature location and traceability recovery. Koschke's approach [8] extracting feature-specific methods is a promising approach to extracting phase-specific methods. Rountev's approach [17] extracting variable names for objects may be effective to extract names of important objetcs representing a phase.

Automated mapping will help developers testing and debugging a program since developers can recognize what they are testing and what the program is doing in terms of features.

Tool Integration. We have integrated the approach into Amida, our sequence diagram visualization tool [24]. Figure 9 is a screenshot of Amida visualizing a sequence diagram of a phase "Maintenance view of a tool". The diagram shows only 40% of method call events in the whole trace. Although we have no prior knowledge on the system, we could filter out the other phases from a sequence diagram.

Our phase detection maybe effectively collaborate with other visualization approaches such as Circular Bundle View [4], feature interaction analysis [11, 18] and hierarchical dynamic slicing [26]. Applying execution reduction [23] before our phase detection is also a promising collaboration to analyze a multi-threaded program.

5. CONCLUSIONS

We proposed a novel approach to efficiently detecting phases, or high-level behavioral units of interest to developers, using a LRU cache for observing a working set of objects. Our algorithm enables developers to investigate a small portion of an execution trace. The approach is lightweight and easy to implement; the technique can collaborate with visualization tools that handle a large execution trace.

We have applied our approach to industrial systems. We found that our approach detects features and their subtasks as phases without the deep knowledge on target applications and parameter settings.

In future work, we would like to investigate a way to automatically map features in a scenario to phases in its execution trace. We are also planning to investigate how the algorithm work in concurrent systems other than enterprise systems. While we are using a fixed-size LRU cache, we are also interested in a cache adaptation approach that is proposed to improve the performance of a system [20].

Acknowledgment

We thank Mr. Ken-ichi Maeda and Mr. Shigeo Hanabusa of Hitachi Systems & Services, Ltd. for supporting our experiment.

This research was supported by Japan Society for the Promotion of Science, Grant-in-Aid for Young Scientists (Start-up) (No.19800021).

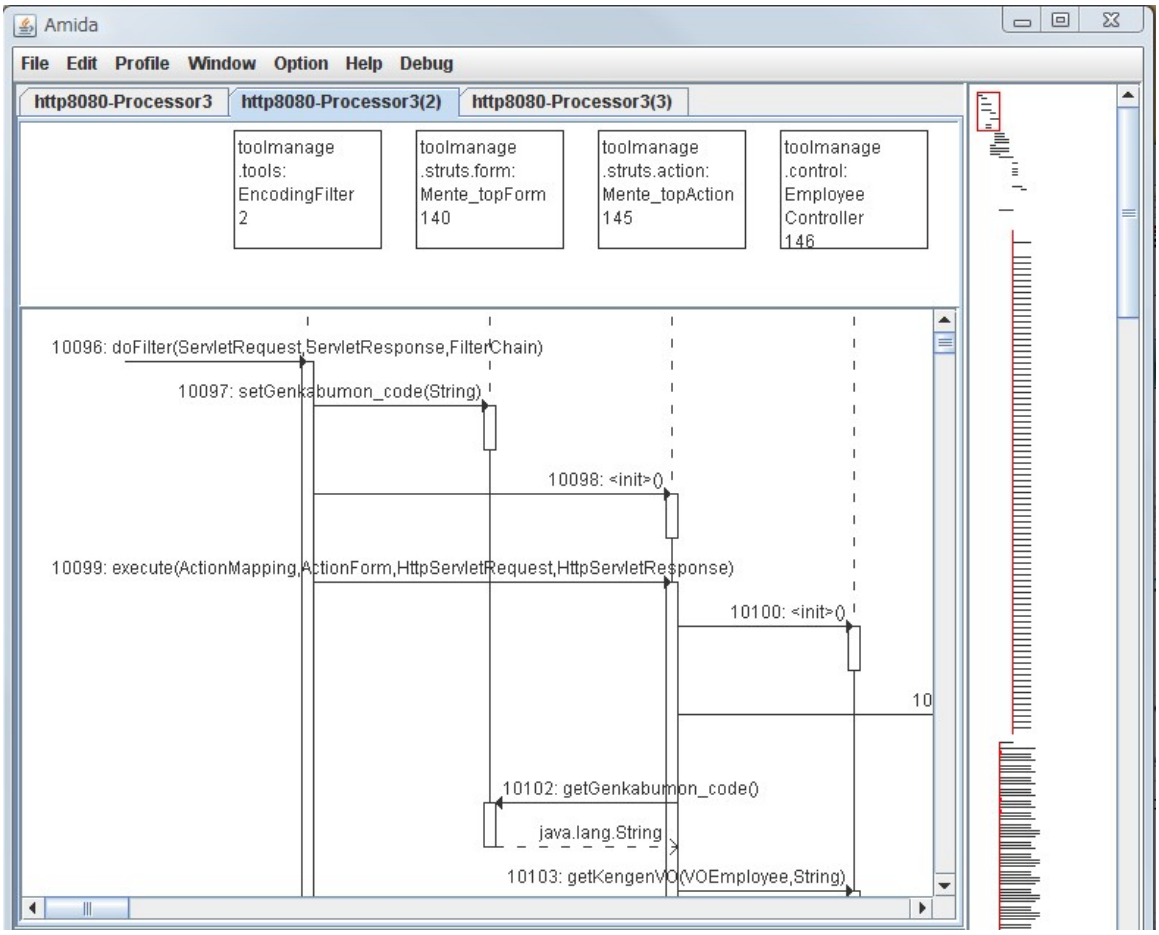


Figure 9: A screenshot of Amida visualizing a phase "Maintenance view of a tool" extracted by our phase detection algorithm.

6. REFERENCES

[1] Briand, L. C., Labiche, Y. and Leduc, J.: Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. IEEE, Vol.32. No.9, pp.642-663, 2006.

[2] Chan, K., Liang, Z. C. L. and Michail, A.: Design Recovery of Interactive Graphical Applications. proc. of ICSE 2003, pp.114-124.

[3] Cornelissen, B., van Deursen, A., Moonen, L. and Zaidman, A.: Visualizing Testsuites to Aid in Software Understanding. Proc. of CSMR, pp.213-222, 2007.

[4] Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J. J. and van Deursen, A.: Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. Proc. of ICPC, pp.49-58, 2007.

[5] Czyz, J. K. and Jayaraman, B.: Declarative and Visual Debugging in Eclipse. Eclipse Technology Exchange, `http://www.cs.mcgill.ca/~martin/etx2007/papers/7.pdf` , 2007.

[6] Eisenbarth, T., Koschke, R. and Simon, D.: Locating Features in Source Code. IEEE Computer, Vol.29, No.3, pp.210-224, 2003.

[7] Hamou-Lhadj, A. and Lethbridge, T.: Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System, Proc. of ICSE, pp.181-190, 2006.

[8] Koschke, R. and Quante, J.: On Dynamic Feature Location. Proc. of ASE, pp.86-95, 2005.

[9] Lau, J., Perelman, E. and Calder, B.: Selecting Software Phase Markers with Code Structure Analysis. Proc. of CGO, pp.135-146, 2006.

[10] Lieberman, H. and Hewitt, C.: A Real-Time Garbage Collector Based on the Lifetimes of Objects. Communications of the ACM, Vol.26, No.6, pp.419-429, 1983.

[11] Lienhard, A., Greevy, O. and Nierstrasz, O.: Tracking Objects to Detect Feature Dependencies. Proc. of ICPC, pp.59-68, 2007.

[12] Nagpurkar, P., Hind, M., Krintz, C., Sweeney, P. F. and Rajan, V. T.: Online Phase Detection Algorithms. proc. of CGO 2006, pp.111-123.

[13] Pauw, W. D., Jensen, E., Mitchell, N. Sevitsky, G., Vlissides, J. M. and Yang, J.: Visualizing the Execution of Java Programs. Revised Lectures on Software Visualization, International Seminar, pp.151-162, 2002.

[14] Reiss, S. P. and Renieris, M.: Encoding Program Executions. Proc. of ICSE, pp.221-230, 2001.

[15] Reiss, S. P.: Dynamic Detection and Visualization of Software Phases. Proc. of WODA, pp.50-55, 2005.

[16] Richner, T. and Ducasse, S.: Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. Proc. of ICSM, pp.34-43, 2002.

[17] Rountev, A. and Connell, B. H.: Object Naming Analysis for Reverse-Engineered Sequence Diagrams. Proc. of ICSE, pp.254-263, 2005.

[18] Salah, M. and Mancoridis, S.: A Hierarchy of Dynamic Software Views. From Object-Interactions to Feature-Interactions. Proc. of ICSM, pp.72-81, 2004.

[19] Sharp, R. and Rountev, A.: Interactive Exploration of UML Sequence Diagrams. Proc. of VISSOFT, pp.8-13, 2005

[20] Shen, X., Zhong, Y. and Ding, C.: Locality Phase Prediction. Proc. of ASPLOS, pp.165-176, 2004.

[21] Shen, X., Shaw, J., Meeker, B. and Ding, C.: Locality Approximation Using Time. Proc. of POPL, pp.55-61, 2007.

[22] Systä, T., Koskimies, K and Müller, H.: Shimba - an Environment for Reverse Engineering Java Software Systems. Software - Practice and Experience, Vol.31, pp.371-394, 2001.

[23] Tallam, S., Tian, C., Zhang, X. and Gupta, R.: Enabling Tracing Of Long-Running Multithreaded Programs Via Dynamic Execution Reduction. Proc. of ISSTA, pp.207-217, 2007.

[24] Taniguchi, K., Ishio, T., Kamiya, T., Kusumoto, S. and Inoue, K.: Extracting Sequence Diagram from Execution Trace of Java Program. Proc. of IWPSE, pp.148-151, 2005.

[25] Ungar, D.: Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. Proc. of the first ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. pp.157-167, 1984.

[26] Wang, Tao. and Roychoudhury, Abhik.: Hierarchical Dynamic Slicing. Proc. of ISSTA, pp.228-238, 2007.

[27] Wilde, N. and Huitt, R.: Maintenance Support for Object-Oriented Programs. IEEE TSE, Vol.18, No.12, pp.1038-1044, 1992.