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 understanding 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, an execution trace may be too large to be transformed into a single diagram. In this paper, we propose a novel approach to efficiently detecting phases, or high-level behavioral units described in a use-case scenario. Our idea is based on the nature of object-oriented programming; a phase starts with preparing objects for the phase and ends with destroying temporary objects. Our technique uses a LRU cache for observing a working set of objects, and interprets a sharp rise in the frequency of the cache update as a phase transition. We have applied our approach to two industrial applications and found that our approach is promising to visualize a phase corresponding to a feature as a sequence diagram.

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, sequence diagram

1. INTRODUCTION

Visualizing collaborations of objects is important for developers understanding 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, 24], and because collaborations of objects provide a larger unit of understanding than classes [14].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WODA – Workshop on Dynamic Analysis, July 21, 2008
Copyright 2008 ACM 978-1-60558-054-8/08/07 ...$5.00.

Many techniques and tools are proposed to visualize dynamic collaborations involved in an execution trace [1, 4, 6, 11, 20, 21].

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 [6, 12]. Another approach is visualizing an overview of a trace using zoom-in/out functionality [11] or a new viewer named Circular Bundle View [3]. The third approach is visualizing only events of interest to developers using a query-based interface provided by JIVE [4] and Shimba [20].

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 [13]. Our method identifies a phase as a consecutive sequence of run-time events. Some phase corresponds to a feature, which is a realized functional requirement of a system [5, 16]. 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. Furthermore, developers can visualize only the login phase as a sequence diagram and investigate its detailed behavior.

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 [8, 22]. We employ a LRU cache for observing objects that are working for the current phase; when the cache is frequently updated, we recognize that a new phase is beginning and preparing objects for the new phase. This lightweight approach is suitable to analyze a large number of events and objects. For example, our prototype without any optimization techniques took only 30 seconds to process an execution trace involving a million events.

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 cover 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 [21]. 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. Developers can read a sequence diagram corresponding to a feature.

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

2. BACKGROUND

Visualizing collaborations of objects is important for developers understanding and debugging an object-oriented program. In general, object-oriented programs are difficult to maintain because of dynamic binding of method calls [24]. While collaborations of objects provide a larger unit of understanding than classes [14], 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, 4, 6, 11, 20, 21]. 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 [6]. Reiss proposes to compress an execution trace into a compact representation [12]. Several visualization tools visualize repeated method calls in a compact style [1, 11, 21].

Another approach is visualizing an overview of a trace. Pauw uses zoom-in/out functionality [11]. Cornelissen proposes a new viewer named Circular Bundle View [3]. 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 [20] and JIVE [4] provide a 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 [17]. Briand developed a tool to visualize only method calls related to Remote Method Invocation in a distributed system [1].

Our phase detection technique divides 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 that corresponds to a feature of interest to the developers. A key difference is that the output of our technique is a sequence of events corresponding to a particular feature. Our technique can collaborate with the previous approaches for visualization, for example, we have integrated the technique to Amida, our sequence diagram visualization tool [21].

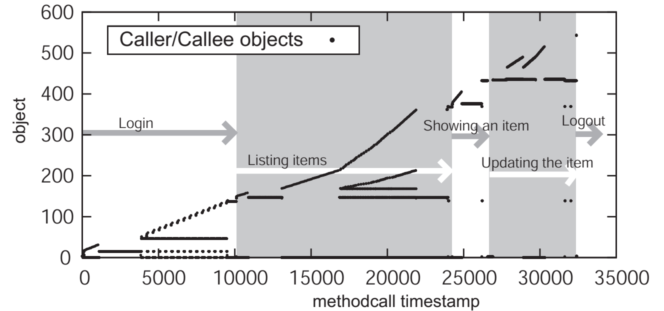

Figure 1: Caller/Callee Objects in a use-case scenario of an industrial system. The execution trace comprises five feature-level phases: login, three steps to update a database record and logout.

We define a phase as a consecutive sequence of run-time events. Some phase corresponds to a feature, which is a realized functional requirement of a system [5, 16]. 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 scenario 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 [11]. 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. The execution trace in the figure comprises five steps in terms of features: login, listing items in a database, editing a selected item, updating the database and logout. Developers can find the features as phases in the figure because different objects are working for each feature. However, it is difficult to manually divide the trace to phases and visualize one of them as a feature-specific sequence diagram. Our phase detection approach enables developers to investigate a small portion of an execution trace corresponding to an interesting feature.

Our phase definition is to map a use-case scenario to an execution trace. Another phase definition based on syntactic structure is proposed by Wang [23]. The definition provides hierarchical phases for developers to investigate the detail of a fault using an execution trace. However, it is hard to handle a method contributing to one or more features with our approach.

Our approach is a kind of feature location [5] 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 [13]) since

```
   :
@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(){
   :
```

Figure 2: A fragment of an execution trace.

performance optimization 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 [10]. Such a model is different from our phase detection.

3. AUTOMATIC PHASE DETECTION

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 [8, 22]. At the beginning of each phase, new objects are created for the new phase and some objects come from the previous phase [9].
- 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`. A phase transition is between a method return and a call, therefore, the depth of the call stack become local-minimum at the beginning of a phase.

We have chosen LRU cache rather than other algorithm since LRU is effectively capture local objects working in a short period [19].

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.

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. All threads share a common timestamp generator in order to serialize all method call events.

calleeID denotes which object is called.

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.

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.

To extract these attributes from a Java program, we are using Amida profiler [21], an implementation of Java Virtual Machine Tool Interface (JVMTI). Figure 2 is a fragment of an execution trace in a textual format that is a part of the trace shown in Figure 1. Each line represents a method call or return event. A call event comprises a thread ID , a timestamp, a method signature, an object ID and a symbol of a method call ({). A return event comprises a thread ID and a symbol of a method return (}). For example, the first line of Figure 2 is a method call event in Thread @1 whose timestamp is 19. The event calls <init> (a constructor) of the LoginForm object whose ID is 5.

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 $threshold$ described later.

An output $P = [t_1, t_2, ..., t_p]$ is a list of timestamps indicating phases in the trace.

The procedure works as follows:

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 time-stamp 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.

This algorithm has four parameters: cache size c, window size w , $threshold$ and phase search distance m .

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.

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.

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. Other three parameters are integer, but threshold is real. A smaller value is more sensitive.

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

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 only 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.

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

We have compared phases detected by our approach with phases manually identified by developers.

4.1 Settings

We have analyzed two Java applications as follows:

Table 1: Execution traces and number of phases manually detected by developers in those traces.

traceID	#events	#objects	#f	#m
T-1	32416	546	5	18
T-2	30494	524	6	19
T-3	26603	438	4	14
T-4	15909	237	3	10
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

- *Tool Management System* is a web application developed by a software industry. This system uses a background thread for managing server resources and three threads for processing HTTP request. We recorded four execution traces corresponding to four use-case scenarios (T-1, T-2, T-3 and T-4 in Table 1). Those scenarios are different from each other, but involve several common features such as login.

- *Book Management System* is a material for a training course of programming used in an industry. This system is also multi-threaded; a thread manages a database in the background while a user operates the system. There are five programs implementing the same specification. We have executed the same scenario on each implementation, and got five traces (L-1 to L-5 in Table 1).

We excluded Java SDK library from the traces, 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 run-time overhead. The execution of a feature in a trace may not be identical with the execution of the same feature in another trace because of the different runtime conditions.

After the execution of the scenarios, we have asked 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 1 shows all execution traces we analyzed; the number of method call events (#events), the number of objects (#objects), the number of manually identified feature-level phases and minor phases (#f and #m) involved in each trace. The use-case of the trace T-1 shown in Figure 1 includes 5 features. The 5 features are divided into 18 minor phases. For example, login phase is achieved by 4 minor phases; showing a login prompt, checking user-name and password, retrieves a list of user's data from database and showing a main interface.

We have applied our method to the execution traces with various parameter settings. 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 involve at most 350

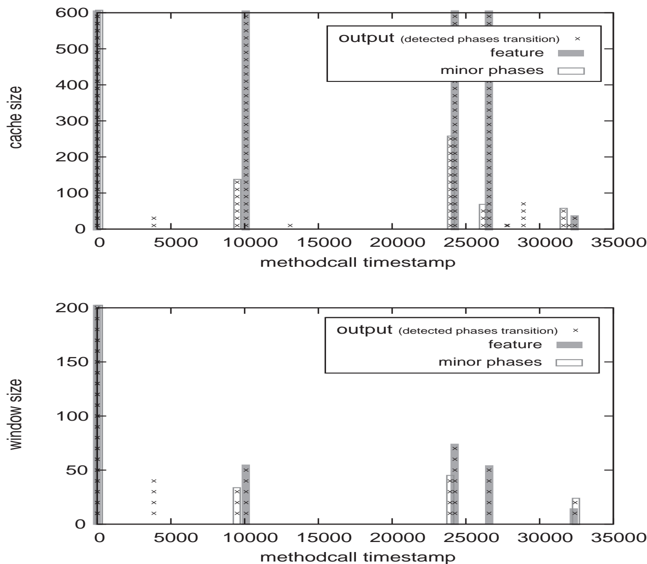

Figure 4: Detected phases in the trace T-1. Window size is fixed ($w = 50$) in the top figure, cache size is fixed ($c = 300$) in the bottom figure.

objects. And we varied window size w from 10 to 300 by 10. We experimentally determined other two parameters: $threshold = 0.1$ and $m = 700$. We took less than five minutes to compute all combination of parameter settings and execution traces on a workstation whose CPU is Xeon 3.0 GHz.

We have compared the detected phases with the manually identified phases. The evaluation is 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

Figure 4 shows effect of cache size c and window size w in the case of T-1. The top figure shows the result from various cache size c and the fixed window size $w = 50$. We plot an x-mark at (x, y) to indicate 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 gray rectangle denotes a minor phase transition event that is also is detected by both of our method and developers. The other x-marks include false positives of our method and phases that are not recognized by developers.

A smaller cache size leads frequent cache update, therefore, our procedure outputs smaller (shorter) phases. A smaller window size also outputs smaller phases since a small window size value regards a small number of new objects as a phase transition. 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. We found that our approach become unstable when a parameter is extremely small (e.g. cache size $c \leq 20$).

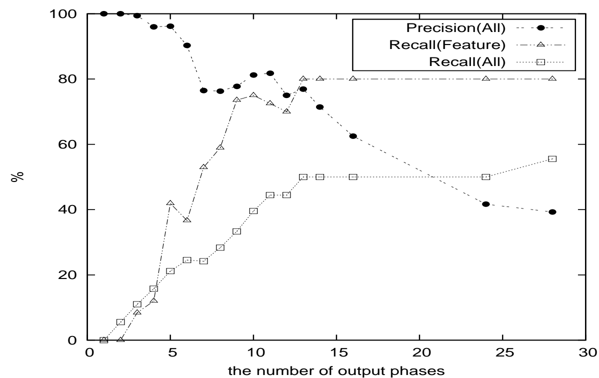

Figure 5: Average recall and precision for various parameter configurations that result in the same number of phases in the trace T-1.

Table 2: Average recall and precision for various parameter configurations that detect 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

4.2.2 Recall and Precision

Figure 5 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 shows high precision when the number of output phases is smaller.

"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. The maximum "Recall(Feature)" is 80% . Our method never detected the logout phase in the trace T-1 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 it is hard to detect a minor phase which is extremely small, e.g. an initialization phase of the feature with our approach.

Table 2 shows the average recall and precision for all possible parameter configurations 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 cover 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 parameter configuration. On the other hand, developers have to estimate the number of phases in an execution trace. This estimation could be done from the number of features in the use-case scenario.

4.2.3 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(L-1 to L-5). Figure 6 shows the phases detected with the same parameters: cache size c = 150 and w = 150. Five vertical arrays of rectangles (L-1 to L-5) represent the execution traces. A rectangle indicates a phase we have detected. The traces include 15 features; horizontal lines indicate the beginning of each feature-level phase manually identified by developers. If the top of a rectangle, or a detected phase, is one of the horizontal lines, the phase precisely corresponds to the beginning of a feature. Otherwise, the top of a rectangle points to a minor-level phase (not a false positive), that is a meaningful step in the use case scenario, in this figure.

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 an important property for developers who modify a program (e.g. in debugging process) since it allows developers to compare traces recorded before and after the modification.

4.3 Discussion

Threats to validity.

Our case study reflects the industrial environment. We have used the industrial systems and use-case scenarios written by the developers of the systems. 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.

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. Based on our experience, we believe this process is not so difficult since the developers could find particular classes and methods indicating a feature. Automating this process is a challenge related to feature location and traceability recovery. Koschke's approach [7] extracting feature-specific methods is a promising approach to extracting phase-specific methods. Rountev's approach [15] extracting variable names for objects may be effective to extract names of important objects representing a phase.

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, but effective to detect phases. We integrated the approach to Amida, our sequence diagram visualization tool. We are preparing to make Amida public in our website [1].

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 works 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 [18].

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).

6. REFERENCES

[1] Briand, L. C., Labiche, Y. and Leduc, J.: Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. IEEE TSE, 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., 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.

[4] 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.

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

[6] 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.

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

[8] Lieberman, H. and Hewitt, C.: A Real-Time Garbage Collector Based on the Lifetimes of Objects.

[1] Amida `http://sel.ist.osaka-u.ac.jp/~ishio/amida/`

Communications of the ACM, Vol.26, No.6, pp.419-429, 1983.

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

[10] Nagpurkar, P., Hind, M., Krintz, C., Sweeney, P. F. and Rajan, V. T.: Online Phase Detection Algorithms. Proc. of Code Generation and Optimization 2006, pp.111-123.

[11] 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.

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

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

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

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

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

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

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

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

[20] 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.

[21] 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.

[22] 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.

[23] Wang, T. and Roychoudhury, A.: Hierarchical Dynamic Slicing. Proc. of ISSTA, pp.228-238, 2007.

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

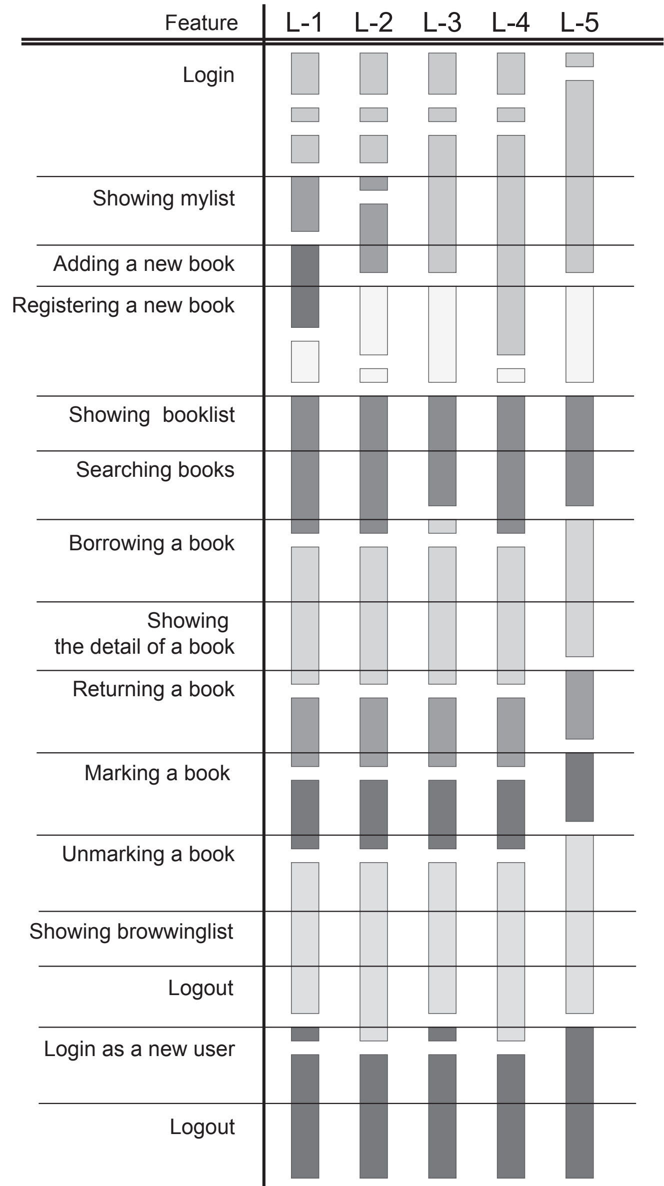

Figure 6: Detected phases in five traces of Library Management Systems with the same parameters.