AMIDA: a Sequence Diagram Extraction Toolkit Supporting Automatic Phase Detection

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

ABSTRACT

Amida is a toolkit to record an execution trace of a Java program and visualize the trace as a sequence diagram. Amida supports our novel approach to efficiently detecting phases; the algorithm precisely divides a long execution trace into a series of smaller diagrams corresponding to features (or tasks to achieve a feature) without deep knowledge on a target system.

Categories and Subject Descriptors

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

General Terms

Experimentation

Keywords

phase detection, software visualization, dynamic analysis, Java

1. INTRODUCTION

An important issue for visualizing an execution trace is how to handle a huge amount of events included in an execution trace. One approach is summarizing an execution trace [3]. Another is visualizing an overview of a trace using zoom-in/out functionality [6] or a new viewer named Circular Bundle View [1].

Our tool, Amida, is a sequence diagram extraction toolkit for Java. Amida implements a *phase detection* approach to dividing a long execution trace into several phases before 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 [7]. 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 [2]. Some other phase may represent a minor phase, or one of the tasks to achieve a feature. Phase detection allows Amida to visualize a feature-level or minor phase as a sequence diagram. This functionality helps developers focus on a small portion of the execution trace that corresponding to a feature or a task that are described in design documents.

2. AMIDA OVERVIEW

Amida is a toolkit for extracting a sequence diagram from an execution trace of Java software. Amida comprises two parts: Amida profiler and Amida viewer. The profiler records an execution trace and the viewer visualize the trace as sequence diagrams.

```
@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 25 String LoginForm(5).getPassword(){
   :

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

         Timestamp                                  Object ID
Thread ID                 Method Signature
```

Figure 1: An example trace

2.1 Amida Profiler

AMIDA profiler is a dynamic link library that implements Java Virtual Machine Tool Interface (JVMTI) to record method call events to be visualized. A user of Amida adds `-agentpath` option to a program execution for connecting the profiler to JVM.

The profiler records three sorts of events: method call, method return and exceptional exit. Each event has the following attributes: timestamp, method signature, object ID and thread ID.

Figure 1 shows an execution log extracted from a web application. Each log line that ends with an open brace ("{") represents a method call event. A close brace ("}") represents a method return from the latest method call in the thread. According to the limited space, we have omitted package names in the trace.

2.2 Amida Viewer

Amida viewer is a graphical user interface for drawing sequence diagrams from trace files generated by Amida profiler. The interface is very simple; the menu [File]-[Load] shows a file open dialog to choose a file to be visualized.

Amida implements a novel phase detection approach to dividing an execution trace into a series of phases. 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 [4]. At the beginning of each phase, new objects are created for the new phase and some objects come from the previous phase [5].
- The beginning and the end of a phase corresponds to a method call and a method return event, respectively.

ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

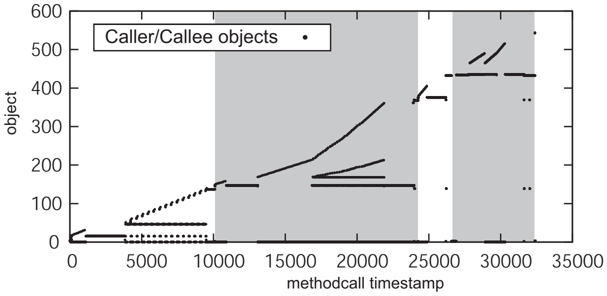

Figure 2: Caller/callee objects in a use-case scenario of an industrial system. The execution trace comprises five feature-level phases: login, three steps (search/show/edit) to update a database record, and logout.

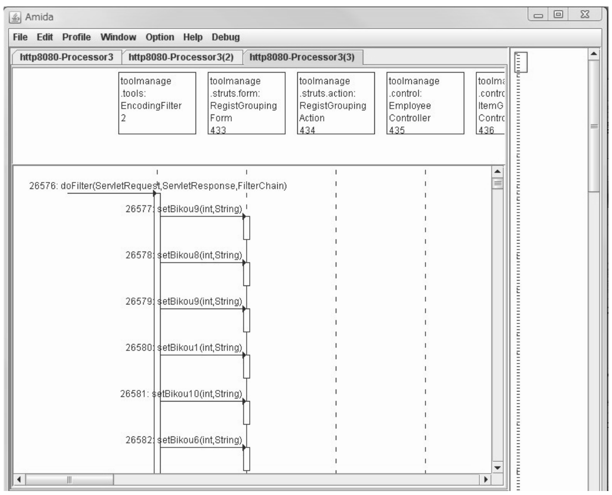

Figure 3: A sequence diagram view of Amida viewer.

Figure 2 shows objects and phases in a use-case scenario we have used in our case study. To automatically detect such phases, we employ a LRU cache for observing the working set of objects.

Amida first prepares an empty LRU cache whose size is c. For each method call event, Amida adds its caller and callee objects to the cache. If the cache is frequently updated in the recent w events, Amida recognizes that a new phase is beginning and creating objects for the new phase. Then Amida identifies a method call event who has the local-minimum depth of the call stack (the latest one if tied) as the beginning of the phase.

After the phase detection, Amida viewer visualizes a portion of the execution trace corresponding to a phase as a compact sequence diagram. Figure 3 is an example screenshot. Amida viewer implements four rules to detect loops and recursive calls that can remove more than 90 percents of the method call for various traces [8].

3. CASE STUDY

We have analyzed execution traces extracted from an industrial web application named "tool management system". The system has JSP pages as user interface, therefore, execution traces include all interaction among JSP pages and business logic objects on the server's JVM.

We have executed 4 use case scenarios in design documents with Amida profiler to record execution traces for each system. We have asked developers of the systems to manually identify phases that represent features in the execution traces. We also asked them to divide a feature-level phase into minor phases that correspond to tasks to achieve the feature. An execution trace involves 4 to 6 features. Each feature comprises 4 minor phases on average.

Table 1: Average recall and precision of phase detection for all settings that detects the same number of phases in the traces.

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

We summarized the result from all traces based on recall and precision. Table 1 shows average recall and precision for all possible parameter configurations that results in 5 or 10 phases. If we got 5 phases with some parameters (arbitrary pair of cache size c and window size w), 93% of them are meaningful for developers (7% are false positives); they covers 56% of the features and 39% of the minor phases in the trace. 10 phases involve 8 correct phases and 2 false phases on average. This result shows that developers can apply our phase detection approach without the deep knowledge on a target system and parameter configuration.

Acknowledgement

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).

4. REFERENCES

[1] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the Int'l Conference on Program Comprehension*, pages 49–58, June 2007.

[2] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. on Softw. Eng.*, pages 210–224, March 2003.

[3] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proceedings of the Int'l Conference on Program Comprehension*, pages 181–190, May 2006.

[4] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, pages 419–429, June 1983.

[5] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings of the Int'l Conference on Program Comprehension*, pages 59–68, June 2007.

[6] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, international Seminar*, pages 151–162, May 2001.

[7] S. P. Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the Int'l Workshop on Dynamic Analysis*, pages 1–6, May 2005.

[8] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. Extracting sequence diagram from execution trace of java program. In *Proceedings of the Int'l Workshop on Principles of Software Evolution*, pages 148–151, September 2005.