LETTER

# Towards Logging Optimization for Dynamic Object Process Graph Construction

**Takashi ISHIO**[†a)], ***Member***, **Hiroki WAKISAKA**[†b)], **Yuki MANABE**[††c)], ***Nonmembers***, ***and*** **Katsuro INOUE**[†d)], ***Fellow***

**SUMMARY** Logging the execution process of a program is a popular activity for practical program understanding. However, understanding the behavior of a program from a complete execution trace is difficult because a system may generate a substantial number of runtime events. To focus on a small subset of runtime events, a dynamic object process graph (DOPG) has been proposed. Although a DOPG can potentially facilitate program understanding, the logging process has not been adapted for DOPGs. If a developer is interested in the behavior of a particular object, only the runtime events related to the object are necessary to construct a DOPG. The vast majority of runtime events in a complete execution trace are irrelevant to the interesting object. This paper analyzes actual DOPGs and reports that a logging tool can be optimized to record only the runtime events related to a particular object specified by a developer.

***key words:*** *dynamic analysis, logging, program understanding, dynamic object process graph*

## 1. Introduction

Logging the execution process of a program is a popular activity for practical program understanding [1]. However, it is challenging to record and analyze a substantial number of runtime events. For example, developers are investigating a web server problem. Developers have to identify the requests from clients that caused the failures from an execution trace, even if the vast majority of events in the trace are related to successful requests.

To focus on a small subset of runtime events which are related to the behavior of a single object, a dynamic object process graph (DOPG) can be used [2], [3]. A DOPG is a partial control-flow graph for a single object, which connects only executed instructions such as branches, method calls, and field access related to the object. Figure 1 shows a source code fragment including three control-flow paths. While its control-flow graph includes 11 nodes, a DOPG for an object created by the shaded path includes 6 nodes as shown on the right side of Figure 1. The other nodes are excluded from the DOPG because they are irrelevant to the object.

Manuscript received January 1, 2013.
Manuscript revised January 1, 2013.

[†]Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka, 565-0871, Japan
[††]Graduate School of Science and Technology, Kumamoto University, 2-39-1 Kurokami, Chuo-ku, Kumamoto 860-8555, Japan
a) E-mail: ishio@ist.osaka-u.ac.jp
b) E-mail: h-wakisk@ist.osaka-u.ac.jp
c) E-mail: y-manabe@cs.kumamoto-u.ac.jp
d) E-mail: inoue@ist.osaka-u.ac.jp
DOI: 10.1587/transinf.E0.D.1

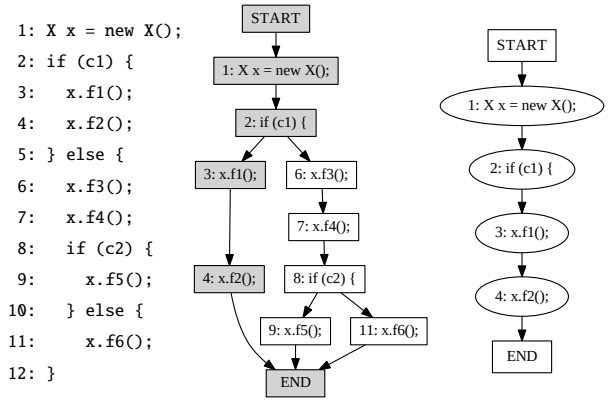


**Fig. 1** A sample source code fragment, its control-flow graph and a DOPG for an object created from the shaded control-flow path.

A developer is often interested in the behavior of a particular object in an execution trace. Although a simple logging tool for DOPG may record all the runtime events and then permit a developer to choose a preferred DOPG, it requires a lot of time and space to record a numerous runtime events and most events are unlikely to be relevant to the interesting behavior. Therefore, this paper discusses an optimization approach for DOPG construction that involves reducing the number of runtime events to be recorded; it requires a developer to specify the method calls for the object of interest. For example, if a developer is interested in only the shaded path in Figure 1, the developer can specify the interesting object that receives a method call at line 3. A logging tool can discard runtime events for objects that reach line 6 and ignore events on paths between line 6 and the end node.

This paper investigates whether actual DOPGs in programs can be specified by method calls and whether a logging tool can be optimized for a DOPG. Because there is no empirical data of DOPGs in literature, the paper analyzes actual DOPGs from a benchmark suite and discusses possible optimizations.

## 2. Analysis

A basic idea for optimization is to identify the objects of interest to a developer. This paper evaluates whether a DOPG can be specified by method calls because a developer will not know the precise shape of a DOPG of interest before

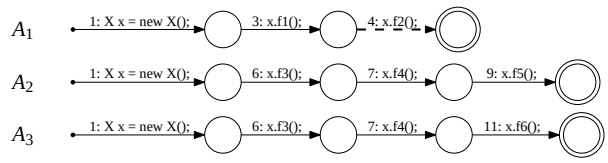


**Fig. 2** DFA accepting method calls on each control-flow path

execution. If a smaller number of method calls are enough to identify an object for a DOPG, a logging tool can ignore additional runtime events that are irrelevant to the DOPG.

A metric for interesting object identification is computed for class $C$ as follows. First, DOPGs are extracted from all instances of class $C$. Each DOPG is translated into a deterministic finite automaton (DFA), which includes only method call events. Figure 2 shows three DFAs corresponding to three DOPGs of control-flow paths in Figure 1. Next, the shortest unique path $l_k$ is computed for each DFA $A_k$. A sequence of method calls is unique to $A_k$ if only $A_k$ has the path from the initial state (the other DFAs reject $l_k$). In Figure 2, the shortest unique paths are indicated by solid edges (10 of 11 edges). To identify an object as relevant to a DOPG, the object must traverse one of the paths. For example, an object was identified as relevant to $A_1$ when it received the method calls at line 1 and line 3. Finally, the metric $R(C)$ for class $C$ is calculated using the following formula:

$$R(C) = \frac{Predict(C)}{Trace(C) + Predict(C)}$$

where $Trace(C)$ is the total number of edges in the paths $l_k$ (solid edges), $Predict(C)$ is the total number of remaining edges (dashed edges). If $R(C)$ is high; a logging tool can ignore more runtime events for irrelevant objects on an average. In case of Figure 2, $R(\mathtt{X}) = \frac{1}{10+1} = 0.09$. This indicates that if three control-flow paths are equally selected, a logging tool has to record most of the events for the class.

To compute the $R(C)$ metric in general programs, actual DOPGs are extracted from five applications in the DaCapo benchmarks: `avrora`, `batik`, `lusearch`, `pmd`, and `xalan`. The applications include 1015 classes in total. The analysis excluded 212 classes, which included at least one object that satisfied one of the following conditions.

- The object has been concurrently accessed by two or more threads. This is because a DOPG has not been defined for a multi-threaded program.
- The object has used recursive calls, because realizable paths of recursive calls cannot be represented by DFA.
- The DFA of the object has more than 50 states. This condition excludes objects globally used in an application since such a class is likely to be a utility class [4].

In Figure 3, $R(C)$ is plotted for each class in a descending order. 535 classes (66% of the analyzed classes) have $R(C) = 1$. Each of these classes is represented by a single DOPG, *i.e.*, each class is used by a particular sequence of method call sites. To construct a DOPG for these classes,

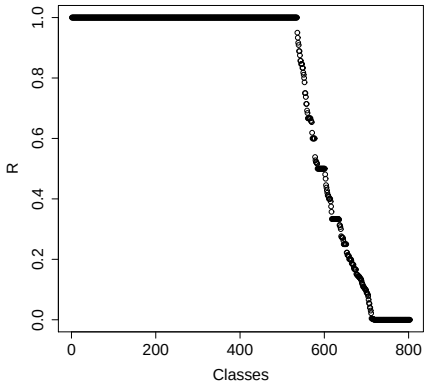


**Fig. 3** $R(C)$ value of 803 classes in descending order.

a logging tool requires only the first single instance created in an execution. A pointer/alias analysis, which identifies source code locations where an object may be used, is also effective for minimizing logging. Many classes are included in this category because a single object is created for a particular feature (or business logic). For example, `DisplayManager` class in `batik` manages display properties of the system. `ClockDomain` class in `avrora` manages a dataset for the whole system.

183 classes (24%) have $0 < R(C) < 1$. Each of these classes has more than two DFAs. The average $Predict(C)$ are 4 method call sites. A logging tool can ignore sections of runtime events for these classes as described in Section 1. In this group, instances are set up in different ways after their instantiation. For example, `StyleSheet` and `PathParser` classes in `batik` are included in this group.

The other 85 classes (10%) have $R(C) = 0$ . Their instances have received the same sequence method calls except for the final method call. Therefore, optimization using method calls are not effective for these classes. `GenericText` class in `batik` is an example of this category.

## 3. Conclusion

This paper analyzed actual DOPGs in terms of the $R(C)$ metric. The result shows optimized logging could be implemented for 90% of the classes. Although the DOPGs used in this paper depend on the DaCapo benchmark, optimized logging for DOPG construction is a promising approach.

Before a new optimized logging tool can be built, two issues need to be addressed. First, the result might miss some DOPGs, because the benchmark does not cover all possible execution paths in the programs. More test cases for improving the path coverage are required to evaluate the effectiveness of the optimization approach more precisely. Secondly, the new logging tool should incorporate a pointer analysis (or an alias analysis) to identify method call sites relevant to interesting objects, because an object can be manipulated by various methods. Furthermore, the effect of the precision of the pointer analysis on the performance of a logging tool should be evaluated.

## Acknowledgement

This work was supported by JSPS KAKENHI Grant Number 23680001.

### References

[1] A. Zeller, Why Programs Fail - A Guide to Systematic Debugging, the Second Edition, Morgan Kaufmann, 2011.
[2] J. Quante and R. Koschke, “Dynamic object process graphs,” Journal of Systems and Software, vol.81, pp.481–501, 2008.
[3] J. Quante, “Do dynamic object process graphs support program understanding?,” Proc. of ICPC, pp.73–82, 2008.
[4] A. Hamou-Lhadj and T. Lethbridge, “Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system,” Proc. of ICPC, pp.181–190, 2006.