Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 481–501
The Journal of Systems and Software
www.elsevier.com/locate/jss

Dynamic object process graphs

Jochen Quante, Rainer Koschke *

Software Engineering Group, Computer Science, Faculty 3, University of Bremen, 28359 Bremen, Germany

Available online 26 June 2007

Abstract

A *trace* is a record of the execution of a computer program, showing the sequence of operations executed. A trace may be obtained through static or dynamic analysis. An *object trace* contains only those operations that relate to a particular object.

Traces can be very large for longer system executions. Moreover, they lack structure because they do not show the control dependencies and completely unfold loops. Object process graphs are a finite concise description of dynamic object traces. They offer the advantage of representing control dependencies and loops explicitly.

This article describes a new technique to extract object process graphs through dynamic analysis and discusses several applications, in particular program understanding and protocol recovery. A case study is described that illustrates and demonstrates use and feasibility of the technique. Finally, statically and dynamically derived object process graphs are compared.

Keywords: Tracing; Protocol recovery; Dynamic analysis; Program understanding; Reverse engineering

1. Introduction

Program slicing is a technique that allows one to identify the statements of the program that influence a variable at a certain program location (Weiser, 1979). It keeps only those statements that are needed to understand how a certain variable, say V, is computed, that is, all statements that compute values directly or indirectly used in the calculation for V (data dependency) and all the statements that decide whether these statements are executed at all (control dependency). Similarly, object process tracing is a technique that identifies all statements that operate on an object of interest and all the statements that decide whether these statements are executed at all plus the flow of control between these statements. The result is an object process graph, that is, a view on the control flow graph from the perspective of a single object. Object process tracing is an enabling technology with applications in many reverse engineering tasks, such as program understanding and protocol recovery.

1.1. Our previous work

Koschke and Zhang (2001) sketched a method for protocol recovery based on object process graphs. Through unifying different usages of a component in a combined graph representation, a first hint of its underlying protocol may be obtained (black-box understanding). This information may be complemented through analyzing the internals of the component, such as explicit checks of preconditions that may raise exceptions (glass-box understanding). Furthermore, the user may validate and enhance the protocol as extracted or may hypothesize a protocol that may then be checked against the actual code (similarly to the idea of the reflexion model in Murphy et al. (1995)).

If a component is a class or abstract data type, it can be instantiated multiple times. Each such instance is an object. A protocol then describes the allowable sequences of operations on every object that is an instance of that component.

* Corresponding author.
E-mail addresses: quante@informatik.uni-bremen.de (J. Quante), koschke@informatik.uni-bremen.de (R. Koschke).
URL: http://www.informatik.uni-bremen.de/st/ (R. Koschke).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.06.005

As a first step toward the underlying protocol, we may extract every sequence of operations on a particular object – essentially every object trace. These object traces may then be unified in the black-box understanding part of protocol recovery. Heiber (2000) investigates how to recover protocols from such static usage patterns. He also discusses different representations of a protocol.

Eisenbarth et al. (2005) describe in detail how object traces can be extracted statically for individual stack and heap objects. They introduce *object process graphs* to represent such traces. Several other researchers use dynamic analysis instead (Ammons et al., 2002; Gschwind and Oberleitner, 2003; Xie and Notkin, 2004; Xie, 2003; Whaley et al., 2002). They typically create recordings of invocations of the component's methods in which they later try to find patterns of execution.

1.2. Contributions

The method described in this article is a stepping stone of our protocol reconstruction undertaking. We describe a new dynamic technique to obtain object process graphs. Object process graphs are finite concise descriptions of object traces. They are essentially sparser control flow graphs that contain only those operations relevant to one object. While dynamic object traces may grow virtually infinitely, object process graphs are limited by the number of nodes and edges in the original control flow graph. We demonstrate that dynamically derived object process graphs can not only serve as a basis for protocol recovery but can also be a good starting point for understanding a program. Also, we compare static and dynamic tracing results to each other.

This paper differs from our earlier paper on this subject (Quante and Koschke, 2006) by additional case studies for program understanding, a discussion on how the technique that was previously described for C can be extended to object-oriented programs, and a quantitative comparison of static and dynamic object process graphs.

1.3. Overview

The remainder of this article is organized as follows. Section 2 describes the concepts of object traces and process graphs. Section 3 describes several potential applications, and Section 4 explains the technique used to extract dynamic object process graphs. Section 5 presents a case study demonstrating use and feasibility of the technique, and Section 6 compares the results of static and dynamic tracing. Section 7 discusses related research.

2. Traces for individual objects

In this section, we define what a trace is in our context and show how sets of traces can be represented by an object process graph. Then, we shortly discuss the differences between static and dynamic tracing.

We start with a motivating example. Consider the C program in Fig. 1. It deals with two stacks `*s1` and `*s2`. We assume the usual semantics for stacks here. Function `read` reads a stack from a file, and `init` creates an empty stack. Although the program has passed all tests, how sure can we be that it does not cause a failure? And in fact, it contains a potential fault that is difficult to spot in the code: a violation of the stack protocol for variable `*s1`. Through the static analysis by Eisenbarth et al. (2005), we can extract all sequences of operations potentially applied to `*s1`.

Each such sequence is a trace. According to Hoare (1985), "a *trace* of the behaviour of a process is a finite sequence of symbols recording the events in which the process has engaged up to some moment in time." While the term *process* is often used in the context of concurrency, it may also be used to denote any kind of object whose behavior is of interest to us. By object, we mean a local or global variable or a variable allocated on the heap at runtime. Hence, we consider an *object trace* a sequence of operations applied to one specific object.

2.1. Object process graphs

The potentially infinite traces can be described in a finite closed form through *object process graphs*. An object process graph (OPG) is a graph:

$$\text{OPG} := (N, E) \quad \text{with} \quad E \subseteq N \times N,$$

where each node $n \in N$ and each edge $e \in E$ can be of one of the following types:

```
01  int main () {
02    int i = 0;
03    Stack *s1 = init ();
04    Stack *s2 = read ();
05    reverse (s2, s1);
06    do {
07      pop(s1);
08      i = i + 1;
09    } while (!empty (s1));
10  }
```

```
11  void reverse
12    (Stack *from, Stack *to)
13  {
14    while (!empty (from)) {
15      push (to, pop (from));
16    }
17  }
```

Fig. 1. Example source code.