Lightweight Control-Flow Abstraction

Christian Lindig Valentin Dallmeier Andreas Zeller
Saarland University
Department of Computer Science
Saarbrücken, Germany
{lindig,dallmeier,zeller}@cs.uni-sb.de

Abstract

To dynamically capture an application's control flow, we propose *call sequence sets* as a light-weight abstraction. Unlike prior approaches based on grammars or trees, call sequence sets are easy to aggregate and compare, which facilitates incremental construction and comparison of control flow within and across program runs.

1 Introduction

What a program does is reflected to a large extend by its control flow (with data flow being another). Control flow is easily captured dynamically as a *trace*—the sequence of statements, basic blocks, or methods being executed. The hard part is the volume of trace data, and the level of detail. Without some abstraction, the volume of data quickly becomes impenetrable (Reiss and Renieris, 2001).

For automatic fault localization in Java applications (Dallmeier et al., 2005) we have captured control flow using *call sequence sets*. They provide a *lightweight*, *compact*, and *abstract* representation with *set semantics*: because of these, we could capture the control flow of individual objects and *aggregate* them into one set per class. Also because of set semantics, we could easily *compare* the control flow of one class across several program runs. We believe that call sequence sets are generally useful to capture and compare control flow.

2 Call Sequence Sets

A *call sequence set* is derived from the trace of methods an object invokes.[1] Such a trace is a long sequence of calls, where each call is characterized by its class, method name, and signature (to resolve overloading).

A call sequence set is obtained from a trace by sliding a window over it. The contents of the window characterize the trace, as demonstrated in Figure 1: a window of width two is slid over a trace of an object that calls `InputStream` (IS) and `OutputStream` (OS) objects. Note that the call sequence ⟨`IS.read`, `OS.skip`⟩ appears twice in the trace, but only once in the call sequence set.

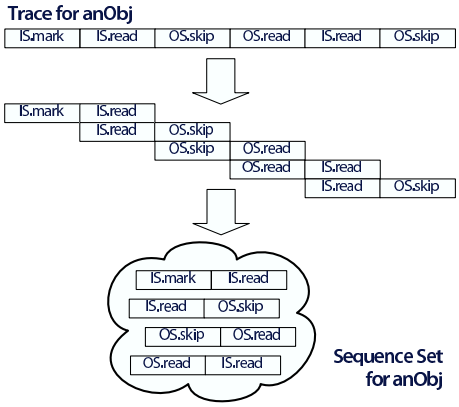

Figure 1: A trace of calls is abstracted to a *call sequence set* using a sliding window of width 2.

Formally, a trace S is a string of calls: $\langle m_1, \ldots, m_n \rangle$. When the window is k calls wide, the set $P(S,k)$ of observed windows are the k-long substrings of S: $P(S,k) = \{w \mid w \text{ is a substring of } S \wedge |w| = k\}$. For example, consider a window of size $k = 2$ slid over S and the resulting set of sequences $P(S,2)$:

$$S = \langle abcabcdc \rangle \quad P(S,2) = \{\langle ab \rangle, \langle bc \rangle, \langle ca \rangle, \langle cd \rangle, \langle dc \rangle\}$$

Going from a trace to its call sequence set is an abstraction that entails a loss of information: different traces may lead to the same call sequence set. The amount to which this happens is controlled by k: larger windows lead to more unique sequences.

The size of a call sequence set may grow exponentially: With n distinct methods, up to n^k different sequences of length k exist. In practice, sequence sets are small, though, because method calls are induced by code, which is static. Its underlying regularity makes a call sequence set a useful and compact abstraction.

2.1 Set Semantics

Because of their set nature, call sequence sets are meaningful to aggregate and compare. For fault localization we traced individual objects and obtained one

[1] This would work equally well for a more fine-grained trace of basic blocks, or any other trace.

call sequence set *per object*. But since objects have no source-code representation, only classes do, we rather liked to characterize classes. We obtained such a characterization by aggregating the call sequence sets of objects into one set *per class*.

Aggregation of call sequence sets is another abstraction that also entails a loss of information. A class' call sequence set may contain call sequences like $\langle ab \rangle$, $\langle bc \rangle$, and $\langle cd \rangle$ from *different* objects. Hence, no single object invoked $\langle abcd \rangle$ in this order, although the class' call sequence set does suggest it. Again, window size k controls the degree to which this happens.

Call sequence sets are meaningful to compare: we used this to compare the behavior of a class in two different program runs. Comparing the respective sequence sets revealed call sequences present in one run, but not the other. A class with many calls present (or absent) in a test-failing run but not in a test-passing run is suspect, and warrants the programmer's attention.

2.2 Fault Localization as an Application

Our fault-location technique ranks classes by comparing method call sequences in passing and failing regression tests (Dallmeier et al., 2005). Classes are ranked high when they exhibit many call sequences that only occur in failing runs. These rankings turned out to perform well in a controlled experiment that used the NanoXML parser as our main subject. In this experiment, we ranked the classes from 33 versions of NanoXML, each with a single known fault.

- In 36% of all test runs, the faulty class was ranked right at the top. On average, a programmer using our technique must inspect 21% of the executed classes before finding the fault. All rankings were noticeably better than random rankings.

- Comparing sequences with $k = 1$ is equivalent to ranking classes based on coverage—another technique for fault localization (Jones et al., 2002). This performed worse than rankings based on sequences with $k \geq 2$ and suggests that call sequences provide useful extra context.

2.3 Implementation and Performance

Our implementation of call sequence sets uses Java bytecode instrumentation (Dahm, 1999) to observe the basic events that form a trace. For efficiency, we compute call sequence sets directly in memory, rather than computing a trace first.

The runtime overhead of tracing varies widely. A computationally intense application—like a ray tracer—that instantiates an extreme number of objects, can be slowed down by a factor of 100 or more. A factor between 10 and 20 is more typical for applications that also perform some I/O operations. The memory overhead is typically below a factor of two, except for applications that instantiate many (small) objects. While the runtime overhead may sound prohibitive, it is comparable to a simpler coverage analysis with JCoverage (Morgan, 2004).

3 Related Work

Reiss and Renieris (2001) survey the representation of control flow as they are used for dynamic analysis. Considerable prior work exists about the compression of traces and call trees, in particular using automata and grammars (Larus, 1999). None of these techniques exhibit set semantics and thus the aggregation and comparison of traces is more difficult than with call sequence sets.

Call sequence sets were inspired by the work of Forrest et al. (1997) about intrusion detection. They use n-grams of system calls to detect abnormal behavior of Unix processes under attack.

4 Conclusions

Call sequence sets are a light-weight data structure to dynamically capture control flow. Their set semantics facilitate an *incremental* or *layered analysis*: sequence sets from sub components (like objects or classes) may be aggregated to represent the control flow of larger entities. Set semantics also facilitate *relative analysis* of control flow: sequence sets from different components or program runs are meaningful to compare.

Call sequence sets generalize the notion of coverage: they indicate the calls executed as well as the temporal context of each invocation.

References

Markus Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, Berlin, Germany, July 07 1999.

Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In Andrew Black, editor, *European Conference on Object-Oriented Programming (ECOOP)*, 2005. To appear.

Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proc. Int. Conf. on Software Engineering (ICSE)*, pages 467–477, Orlando, Florida, May 2002.

James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 259–269, Atlanta, Georgia, May 1–4, 1999.

Peter Morgan. JCoverage 1.0.5 GPL, 2004. URL `http://www.jcoverage.com/`.

Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proc. of the 23rd Int. Conf. on Software Engeneering (ICSE)*, pages 221–232, Los Alamitos, California, May12–19 2001. IEEE Computer Society.