Dynamically Inferring, Refining, and Checking API Usage Protocols

Michael Pradel

Laboratory for Software Technology
ETH Zurich, Switzerland

Abstract

Using a set of API methods often requires compliance with a protocol, whose violation can lead to errors in the program. However, most APIs lack explicit and formal definitions of these protocols. We propose a dynamic program analysis for automatically inferring and refining specifications of correct method call sequences. Our experiments with several Java programs show that we can infer meaningful protocols, such as widely respected programming rules. Furthermore, our analysis finds violations of the inferred specifications that point out potential bugs to the programmer.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Reliability, Verification

Keywords Specification mining, Runtime verification

1. Introduction

Formal specifications of correct method call sequences on application programming interfaces (APIs) are useful for finding bugs, verifying the correctness of a program and also serve as documentation. Since no such *API usage protocols* exist for many existing libraries and frameworks, specification mining techniques, which infer legal method call sequences from runtime data of a program, have been proposed [Ammons et al. 2002, Yang et al. 2006, Gabel and Su 2008]. Unfortunately, inferred specifications are often erroneous and incomplete. This can happen because method calls that are relevant for one specification are interleaved with irrelevant calls, or because the analyzed program execution does not fully utilize the API. Another limitation of many existing techniques is to focus on method calls on single classes or objects. As a result, they miss more complex API usages that encompass multiple related objects.

Our approach addresses these problems by (i) iteratively refining specifications based on several executions of programs using a particular API and (ii) considering protocols involving multiple related objects.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
ACM 978-1-60558-768-4/09/10.

2. Approach

We propose a dynamic analysis that infers, refines, and checks API usage protocols in two phases: The *inference phase* derives finite state machines (FSMs) that describe legal method call sequences, such as Figure 1. In the *checking and refinement phase* the analysis detects confirmations and violations of the inferred protocols; it uses them to refine the specifications and to report potential bugs.

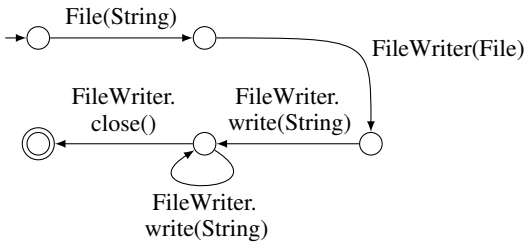

Figure 1. An inferred object usage protocol.

2.1 Inferring specifications

The first phase of the analysis infers protocols from method traces [Pradel and Gross 2009], which we obtain by instrumenting and running an application that uses the API. The inserted instructions report for each method call and return the signature of the invoked method, as well as the object identity and type of caller, callee, arguments, and return value of the call. Instead of analyzing these data as a whole, we transform it into small episodes of related calls that can be analyzed separately. Roughly, each such an episode consists of the methods called during the execution of a single method.

To eliminate method calls that are not relevant for a particular specification, we further filter the call sequences: First, we observe that related method calls are linked by a dataflow relation. For instance, $m_1()$ and $m_2()$ are linked if $m_2()$ is invoked on the return value of $m_1()$. Second, related calls often belong to classes in the same package. Based on these two observations, we extract call sequences in which all calls are dataflow-related and belong to the same package.

Similar call sequences lead us to FSMs that describe typical usage protocols. Two call sequences are considered to be similar if the same sets of methods are called on the involved objects. The order of calls can be different, though. To construct FSMs, we map each method to a state and create a transition whenever two calls are observed consecutively. Each transition is labeled with the method signature of the state that it points to (see Figure 1).

2.2 Checking and refining specifications

Inferred specifications can be used to verify at runtime whether an application conforms to the detected rules. We perform such runtime monitoring using similar episodes of related calls as in the inference phase (Section 2.1). If a call sequence matches parts of a protocol, we analyze whether the calls conform to the specification and report a confirmation or a violation. For example, a method that creates a *FileWriter* and writes to it triggers the protocol in Figure 1. If the programmer forgot to call *close()* at the end, though, a violation is reported.

Dynamic specification inference can derive only API usage protocols that actually occur in the analyzed execution of a program. Hence, legal method calls may be missing. In Figure 1, for instance, calls to *FileWriter.flush()* should be permitted before closing the writer. However, a program execution calling *flush()* would trigger a violation of the protocol in Figure 1. Another source of false positives are transitions resulting from incidental method calls that are not necessarily part of the specification. For example, a call to *File.canWrite()* may precede creating the *FileWriter* but is not required.

Our approach to correct incomplete and erroneous specifications is to exploit the results from runtime monitoring to iteratively refine the inferred specifications. Each new program execution produces a list of confirmations and violations. A specification with many confirmations and several violations resulting from a missing transition in a particular state is likely to be correct, but must be extended with the missing transition. In contrast, a specification with various violations is likely to be the result of an incidental call sequence in the initially analyzed program execution and therefore should be discarded. In addition to refining protocols, we also remove erroneous ones by simply pruning all specifications whose confirmation/violation ratio is below a certain threshold.

3. Experiments

We implemented our approach and evaluated it with real-world Java applications. The following reports on a case study on inferring usage protocols for objects from the Java standard library. We analyzed method traces from six applications: *Eclipse*, *PMD*, and *ANTLR* (all as part of the DaCapo benchmark suite [Blackburn et al. 2006]); *jEdit*, [1] *JabRef*, [2] and *XMLUnit*. [3] Overall, we analyzed around three million runtime events.

The method traces from four of the six programs were given as input to the inference phase. Focusing on API calls to the Java standard library, 131 specifications were generated. Using these protocols, we checked traces from three applications and pruned all protocols with more violations than confirmations. 44 specifications remain, out of which 22 come from two or more distinct call sites in the source code. The results suggest that our inference technique, combined with a simple pruning, produces a significant amount of typical usage protocols, rather than incidental call sequences. A larger case study is reported on in [Pradel and Gross 2009].

To illustrate the effect of our refinement technique, consider a protocol of *StringBuffer*, inferred from *ANTLR*. Initially, it permits to append *String*s and *int*s to the buffer, followed by a call of *toString()*. Only 52 % of the checked *StringBuffer* uses follow this protocol. In a first refinement step, we also permit appending other objects (*char*, etc.) and calling *toString()* right after creating the buffer. This increases the confirmation rate to 70 %. In a second step, we also permit calls to *StringBuffer.length()*, raising the confirmation rate to 74 %. All the refinement steps were inferred from violations of the protocol.

Violations of protocols are, besides being useful to refine specifications, indicators of potential bugs. For instance, we detected a bug using the protocol in Figure 1. A method in the *XMLUnit* project initializes a *FileWriter* without closing it at the end of the method or passing it to another method for doing so. The protocol in Figure 1 was initially found in an execution of *JabRef*. The example illustrates that our approach is able to automatically infer common programming practices ("close all writers that you open") and detect violations of them.

4. Related Work

Ernst et al. present techniques for dynamically discovering program invariants [Ernst 2000]. Ammons et al. mine API usage protocols from method traces with a probabilistic FSM learner [Ammons et al. 2002]. Gabel et al. enhances the performance of such an approach by focusing on predefined micro-patterns [Gabel and Su 2008]. Alternatively, correct method call sequences can also be inferred statically [Whaley et al. 2002]. As an application of our results, we are currently investigating how to check the protocols that our analysis generates with existing static and dynamic verification techniques, such as type state checking [Bierhoff and Aldrich 2007] and runtime monitoring frameworks [Chen and Rosu 2007].

5. Concluding Remarks

This work contributes by considering protocols of multiple related objects and refining its results iteratively. It is therefore a further step to make API usage protocols accessible without the need to write them manually.

Acknowledgments

Special thanks to my PhD advisor Thomas R. Gross.

References

Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL*, 2002.

Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, 2007.

Stephen M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

Feng Chen and Grigore Rosu. MOP: An efficient and generic runtime verification framework. In *OOPSLA*, 2007.

Michael Dean Ernst. *Dynamically discovering likely program invariants*. PhD thesis, University of Washington, 2000.

Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *FSE*, 2008.

Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE*, 2009.

John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, 2002.

Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.

[1] `http://jedit.org`

[2] `http://jabref.sf.net`

[3] `http://xmlunit.sf.net`