Dynamically Inferring, Refining, and Checking API Usage Protocols

Michael Pradel

Laboratory for Software Technology, ETH Zurich

Advisor: Thomas R. Gross

ACM membership number: 5276724

Problem and Motivation

Most real-world programs build upon existing software components, such as libraries and frameworks. To access them programmers typically use an application programming interface (API). While using an API can greatly increase programmer productivity, incorrect API usage is also an additional source of programming errors.

A particular difficulty when programming against an API is that many APIs require programmers to follow certain *API usage protocols*. An API usage protocol is a formal specification that describes in which order the methods of an API may be legally called. One way to represent an API usage protocol is as a finite state machine (FSM), where transitions are labeled with method calls. For example, using an input stream from the Java I/O API, a programmer must at first create the stream, then can read from the stream a certain number of times, and eventually must close the stream. Violating such protocols leads to programming errors. For example, unclosed input streams can lead to resource leakage, and as a result, to a crash of the program.

Although most APIs impose constraints on the order in which programmers use API methods, the corresponding usage protocols are usually not formally specified. To address this problem, dynamic specification mining has been proposed [2,9,7], which infers FSMs describing legal method call sequences from existing programs that use an API. 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.

Our research addresses the problem of inferring accurate API usage errors In a nutshell, our approach is to address both inference and checking of protocols questions together, which helps both in inferring better API usage protocols and in finding more API usage errors.

As a concrete example, consider the source code in Figure 1, which is a simplified version of a method found using our approach in a real-world Java project. The method uses a *FileWriter* to read data from a file, but does not close the writer at the end of the method. An implementation of our approach infers the protocol in Figure 2, which describes correct usage a *FileWriter*. Furthermore, our analysis uses the protocol to find the bug in Figure 1.

```
private void testXMLWithDTD_() throws Exception {
  File tempDtdFile = new File(getName());
  tempDtdFile.deleteOnExit();
  FileWriter dtdWriter = new FileWriter(tempDtdFile);
  dtdWriter.write(aDTD);
  try {
    // ...
  } finally {
    tempDtdFile.delete();
  }
  // dtdWriter is still open!!!
}
```

Figure 1: API usage error: The *FileWriter* needs to be closed after using it.

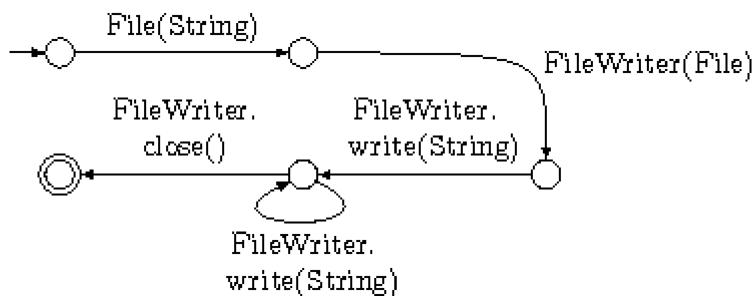

Figure 2: An inferred object usage protocol.

Background and Related Work

Several approaches to mine finite state machines (FSMs) describing legal method call sequences have been proposed. Ammons et al. pioneered to mine specifications of method call sequences from dynamic execution traces using a probabilistic FSM learner [2]. SMArTIC is a specification mining framework that structures the mining process into trace filtering, trace clustering, probabilistic FSM learning, and merging of FSMs [9]. Javert mines FSMs describing typical method call sequences by first mining traces for a set of pre-defined micro-patterns, and then merging them into larger FSMs [7]. Alternatively to dynamically inferring specifications, correct method call sequences can also be inferred statically [12].

Unfortunately, existing approaches do not sufficiently solve the problem of incorrect and incomplete specifications. As a result, using inferred protocols for automated error detection leads to large numbers of false positives. 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.

To find violations of API usage protocols, we can build upon existing static and dynamic verification techniques, such as type state checking [3], framework interaction analysis [8], and runtime monitoring frameworks [1,6,5].

Uniqueness of our Approach

Our work addresses the problem of erroneous and incomplete mined specifications by iteratively refining specifications based on several executions of programs using a particular API. To the best of our knowledge, we are the first to integrate inference, refinement, and checking of API usage protocols into a single 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 2. In the *checking and refinement phase* the analysis detects confirmations and violations of the inferred protocols and uses them to refine the specifications and to report potential bugs.

Inferring specifications

The first phase of the analysis, described in detail in [10] and [11], infers protocols from dynamic method traces. By method trace we mean a sequence of method call and method return events observed in a running program. We obtain method traces by instrumenting and running a program 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 extract *object collaborations*, that is, subsequences of related method calls on sets of related objects. Roughly, each object collaboration consists of the methods called during the execution of a single method and the objects on which these methods are called.

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, *m1()* and *m2()* are linked if

m2() is invoked on the return value of *m1()*. Second, related calls often belong to classes in the same package. Based on these two observations, we filter object collaborations to keep only calls that are dataflow-related and belong to the same package.

To obtain API usage protocols, our analysis summarizes similar object collaborations into FSMs. Two object collaborations 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. Transition weights indicate how often each transition was observed in the traces. Weights can be used to prune incidental call sequences, which have smaller weights in general.

Checking and refining specifications

Inferred specifications can be used to verify at runtime whether a program conforms to the detected rules. We perform runtime monitoring using similar sequences of related calls as in the inference phase (Section 3.1). 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 2. If the programmer forgot to call *close()* at the end, though, a violation is reported.

Dynamic specification inference can only derive API usage protocols that actually occur in the analyzed execution of a program. Hence, legal method calls may be missing. In Figure 2, 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 2. 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.lastModified()* may precede creating the *FileWriter* but should not be part of the API usage protocol.

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 pruning all specifications whose confirmation/violation ratio is below a certain threshold.

Results and Contributions

We implemented our approach and evaluated it with real-world Java programs. In the following, we report results on inferring API usage protocols and preliminary results on automatically refining and checking protocols. Full details on the evaluation of the inference part of our analysis have been published in [10] and [11].

The evaluation of our protocol inference approach addresses two questions: First, does our analysis infer typical API usage protocol? Second, does the analysis scale to large method traces as they result from executing real-world programs? We analyze 240 million runtime events generated by executing ten programs of the DaCapo benchmark suite [4] and infer API usage protocols for the Java standard library.

To assess how representative the inferred protocols are, we compare the results from different programs for the *java.util* API, which is used by all programs. We consider a protocol to be *confirmed*, if it is found independently in more than one program. If two identical protocols are found, both are *confirmed by identity*. If a protocol is included in a protocol found in another program, the first is *confirmed by inclusion*. Table 1 shows the proportions of confirmed and unconfirmed protocols inferred during our experiments. More than a third of the protocols (35%) are confirmed. The majority of the confirmed FSMs appear identically in different result sets. We conclude that a significant part of the specifications that our approach infers are typical object usage protocols.

Table 1: Proportion of confirmed and unconfirmed specifications inferred for the *java.util* package.

Program	FSMs	Confirmed		
		Identity	Inclusion	Overall
antlr	12	7	2	75%
chart	7	1	1	29%
eclipse	87	8	8	18%
fop	39	9	8	44%
hsqldb	1	1	0	100%
jython	34	6	9	44%
luindex	12	5	2	58%
lusearch	3	3	0	100%

pmd	20	2	4	30%
xalan	32	5	6	34%
Overall	247	47	40	35%

To evaluate whether our analysis scales to real-world programs, we measure the time required to analyze large method traces. Figure 3 depicts the execution times of the analysis for ten benchmark programs. The execution times range between 7.8 seconds (*pmd*) and almost 30 minutes (*xalan*). The result show that the runtime of our protocol inference algorithm scales linearly with the number of runtime events to analyze. The main reason for the scalability of the analysis is that each object collaboration can be analyzed separately, without knowledge about prior or future object collaborations.

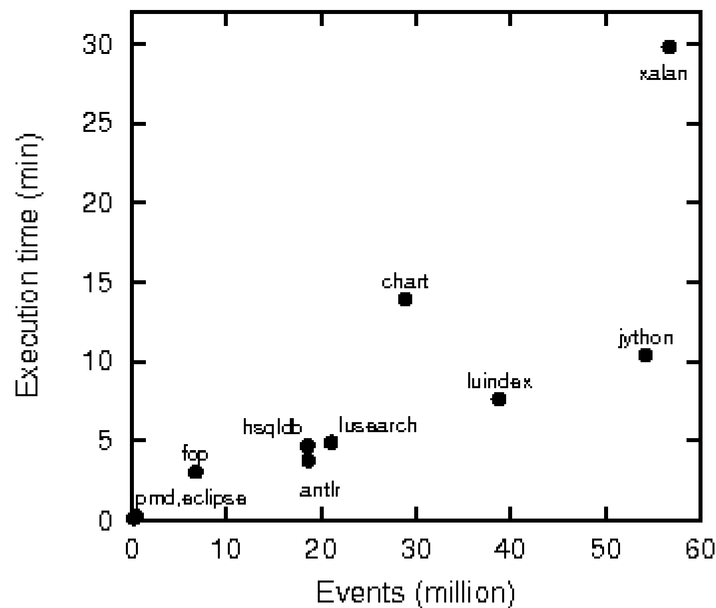

Figure 3: Execution time of the analysis against the number of analyzed events.

To evaluate our approach for iteratively refining and checking API usage protocols, we perform a second case study involving parts of the DaCapo benchmarks and three other programs (*jEdit*, *JabRef*, and *XMLUnit*). Overall, we analyze around three million runtime events in this second case study.

The method traces from four of the six programs are given as input to the inference phase. Focusing on API calls to the Java standard library, 131 specifications are generated. Using these protocols, we check traces from three programs and prune 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.

more distinct call sites in the source code.

To illustrate the effect of our refinement technique, consider a protocol of *StringBuffer* (Figure 4), which was 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.

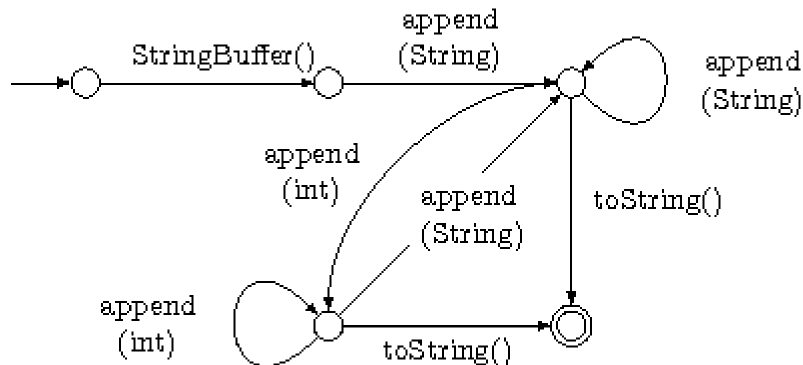

Figure 4: An inferred object usage protocol.

Violations of protocols are, besides being useful to refine specifications, indicators of potential bugs. For instance, we detect a bug using the protocol in Figure 2. A method in the *XMLUnit* project (Figure 1) initializes a *FileWriter* without closing it at the end of the method or passing it to another method for doing so. 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.

Conclusion and Ongoing Work

We present a novel approach to dynamically infer, refine, and check API usage protocols. To the best of our knowledge, we are the first to integrate these three aspects into a single technique. As a result, our analysis infers typical API usage protocols, refines them to account for incomplete and imprecise protocols, and detects erroneous usages of APIs in the analyzed programs.

The refinement and checking part of our analysis is subject of ongoing work. We are currently integrating it into a sophisticated runtime monitoring framework [6] and plan to apply the approach in a larger case study.

Acknowledgments

Special thanks to my PhD advisor Thomas R. Gross.

Bibliography

1

Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble.
Adding trace matching with free variables to AspectJ.
In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345-364. ACM, 2005.

2

Glenn Ammons, Rastislav Bodík, and James R. Larus.
Mining specifications.
In *Symposium on Principles of Programming Languages (POPL)*, pages 4-16. ACM, 2002.

3

Kevin Bierhoff and Jonathan Aldrich.
Modular typestate checking of aliased objects.
In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301-320. ACM, 2007.

4

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann.
The DaCapo benchmarks: Java benchmarking development and analysis.
In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169-190. ACM, 2006.

5

Eric Bodden.
Efficient hybrid typestate analysis by determining continuation-equivalent states.
In *International Conference of Software Engineering (ICSE)*. ACM, 2010.

6

Feng Chen and Grigore Rosu.
MOP: An efficient and generic runtime verification framework.
In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 569-588. ACM, 2007.

7
Mark Gabel and Zhendong Su.
Javert: Fully automatic mining of general temporal properties from dynamic traces.
In *Symposium on Foundations of Software Engineering (FSE)*, pages 339-349. ACM, 2008.

8
Ciera Jaspan and Jonathan Aldrich.
Checking framework interactions with relationships.
In *European Conference on Object Oriented Programming*, 2009.

9
David Lo and Siau-Cheng Khoo.
SMArTIC: Towards building an accurate, robust and scalable specification miner.
In *Symposium on Foundations of Software Engineering (FSE)*, pages 265-275, 2006.

10
Michael Pradel and Thomas R. Gross.
Automatic generation of object usage specifications from large method traces.
In *International Conference on Automated Software Engineering (ASE)*, pages 371-382, 2009.

11
Michael Pradel and Thomas R. Gross.
Mining API usage protocols from large method traces.
In David Lo, Khoo Siau Cheng, Jiawei Han, and Chao Liu, editors, *Mining Software Specifications: Methodologies and Applications*. CRC Press, 2011.
To appear.

12
John Whaley, Michael C. Martin, and Monica S. Lam.
Automatic extraction of object-oriented component interfaces.
In *Symposium on Software Testing and Analysis (ISSTA)*, pages 218-228. ACM, 2002.