Automatic Generation of Object Usage Specifications from Large Method Traces

Michael Pradel
Laboratory for Software Technology
Department of Computer Science
ETH Zurich, Switzerland

Thomas R. Gross
Laboratory for Software Technology
Department of Computer Science
ETH Zurich, Switzerland

***Abstract*—Formal specifications are used to identify programming errors, verify the correctness of programs, and as documentation. Unfortunately, producing them is error-prone and time-consuming, so they are rarely used in practice. Inferring specifications from a running application is a promising solution. However, to be practical, such an approach requires special techniques to treat large amounts of runtime data. We present a scalable dynamic analysis that infers specifications of correct method call sequences on multiple related objects. It preprocesses method traces to identify small sets of related objects and method calls which can be analyzed separately. We implemented our approach and applied the analysis to eleven real-world applications and more than 240 million runtime events. The experiments show the scalability of our approach. Moreover, the generated specifications describe correct and typical behavior, and match existing API usage documentation.**

***Keywords*-Specification inference, dynamic analysis, formal specifications, temporal properties**

I. Introduction

Typical object-oriented applications involve a large number of objects that interact by invoking each other's methods. In general, not every possible call sequence is legal. Instead, rules exist that restrict which method calls are allowed in a particular situation. Unfortunately, such rules are often implicit in the source code and only known to its developers. In particular, users of application programming interfaces (APIs) often have difficulties to understand how the API should be used, because precise and up to date descriptions of typical and correct usage are missing.

Formal specifications of correct method call sequences make legal object interactions explicit and machine-processable. As a result, one can formally prove the absence of certain errors [1], [2]. Furthermore, program analysis tools use specifications to identify potential errors and unusual programming patterns [1], [3], [4], [5], [6]. Specifications are also documentation artifacts, which can be used to understand software written by others [7]. Despite these benefits, formal specifications are rarely used in practice, since writing them is cumbersome, time-consuming, and requires detailed knowledge of the program to specify.

This paper presents a novel technique for the automated generation of specifications of legal method call sequences on multiple related objects. The proposed analysis produces specifications, which ideally should have been written before the implementation, afterwards by analyzing common usages of an API by existing programs. An automated approach is advantageous because it allows non-experts to extract up to date rules of an arbitrary program.

Our analysis derives specifications from method traces of running programs. A dynamic analysis has the advantage that correct execution paths can be distinguished from incorrect paths, based on the assumption that frequent behavior is presumably correct [8]. Moreover, we can use precise type and aliasing information and do not have to rely on approximations of possible behavior.

A major challenge is the large volume of runtime traces. Our approach is to focus on small sets of objects and method calls, called *object collaborations*, that are related because they are used together during the execution of a single method. Each object collaboration can be analyzed separately, without considering other object collaborations. As a result, our analysis executes in reasonable time, even for millions of events.

Our analysis produces finite state machines (FSMs) that describe legal sequences of method calls. Figure 1 shows a FSM that was generated during our experiments. It contains two objects of type *Iterable* and *Iterator* from the Java standard library. The FSM describes how to use an iterator correctly: every call to *next()* should be preceded by a call to *hasNext()*. The weights show how often each transition was observed and therefore indicate how common a particular usage is.

Experiments with an implementation of our approach show its scalability and the quality of the derived specifications. We applied the analysis to eleven Java applications, producing more than 240 million runtime events overall. Even the largest method traces took less than 30 minutes to analyze. Furthermore, we evaluate whether the resulting FSMs describe characteristic behavior rather than incidental call sequences. We find that 35% of all specifications produced for the *java.util* package describe common behavior

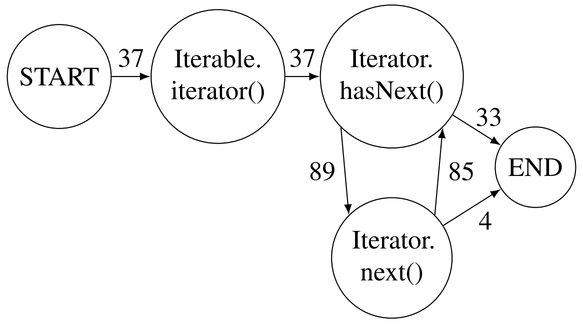


Figure 1. A generated finite state machine. It describes correct iterator usage in Java. A transition between two states indicates method calls which may occur consecutively. The weights report how often a transition was observed.

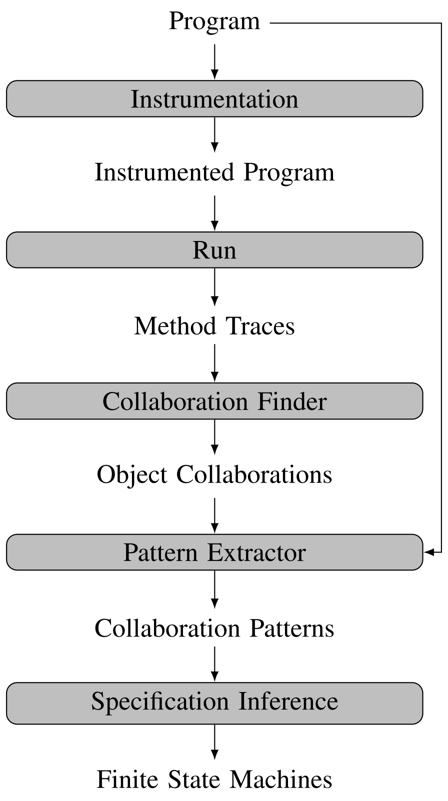


Figure 2. The specification generation system.

that appears in more than one application. Finally, we show that our approach can recover existing specifications of legal method call sequences that are given in textual form, and hence, is able to produce API usage documentation.

Previous work uses static analyses [9], [4], [6] and dynamic analyses [8], [10] to infer rules of program behavior. Many existing approaches focus on non-object-oriented programming languages [8], [4] or on legal call sequences on single objects [9]. Wasylkowski et al. [6] and Gabel et al. [10] generate specifications involving multiple objects using a static and a dynamic approach, respectively. Static approaches can only give general estimations of correct behavior due to the lack of precise type and aliasing information. Our approach is complementary to that of Gabel et al. They consider all method traces of an application as input to a language learning algorithm and reduce complexity by focusing on predefined specification templates. In contrast, we preprocess method traces to extract small sets of related runtime events, which can be analyzed separately.

This paper makes the following contributions:

- **Reduced complexity.** We focus on dynamic object collaborations, a construct that combines related objects and method calls. This strategy reduces the complexity of analyzing runtime traces, because each object collaboration can be analyzed separately.
- **Mining techniques.** We identify similar execution sequences using a set of novel mining techniques, such as comparing the *roles* that objects play within a certain context.
- **Specifications of interacting objects.** We derive finite state machines describing legal method call sequences on multiple objects.
- **Implementation and experiments.** Our results suggest better performance and scalability than previous approaches. We validate the inferred specifications by comparing the results from several applications that use the same library and by comparing the specifications to existing textual documentation.

The following section explains our specification inference system. Section III describes our implementation, which we use for an experimental validation (Section IV). A discussion of our results follows in Section V. Finally, Sections VI and VII relate this paper to existing work and conclude.

II. Approach

This section details our approach and explains the different steps we take to infer specifications of legal method call sequences on multiple related objects from a running program. Figure 2 gives an overview of the system. We instrument programs so that executing them produces a trace of method calls. These traces are the input of our analysis. As a first analysis step, we preprocess the stream of runtime events to extract sets of collaborating objects (object collaborations). By summarizing similar object collaborations into recurring patterns (collaboration patterns) we obtain a variety of similar method call sequences, from which we finally derive temporal specifications. Our approach uses runtime data to decide how common various call sequences are and infers correctness from common uses.

A. Instrumentation and Run

We instrument programs in such a way that runtime events are written into a log file during the program's execution. Specifically, the instrumented program reports the following for each method call and return: unique object identifier and dynamic type of caller and callee, name and parameter types of the called method, and the source code location of the call. The source code location of a method call is required to distinguish between different call sites within one method. Static method calls are ignored, as our analysis focuses on related objects and static calls cannot be mapped to any object. Exceptional control flows are not handled in a special way. If a method throws an exception, we handle the method as if it would return and report the corresponding return event.

The analysis is performed offline, that is, after running the application. This setup provides the benefit that event log files of different applications can be combined, for instance, to derive specifications of a library used by multiple applications. Moreover, the analysis can be performed independently from the program run.

Inferring specifications from runtime data, rather than source code, has the benefits of precise type and aliasing information and does not require any approximations of real behavior. However, deriving reasonably complete specifications is difficult in practice, since one requires test cases that exercise as many execution paths as possible. Also, we cannot guarantee that all generated specifications describe semantically correct behavior, because the input data may contain illegal call sequences that did not lead to a program crash for some reason. We face the first problem by using large volumes of method traces that are derived from unit tests and benchmark suites, giving us an extensive and reproducible starting point for our analysis. To approach the correctness problem we exploit an observation from Ammons et al. [8], which is that execution frequencies can be used as an estimate for the correctness of observed behavior.

B. Object Collaborations

Running an instrumented program yields a large volume of method traces. This entails two major challenges for deriving legal execution patterns from such data. On the one hand, we must focus our attention on sets of related method calls. A naive approach would be to combine arbitrary method calls that appear somewhere in the traces. However, many useless specifications would result from method calls without any semantic link. On the other hand, we need to ensure the scalability of our analysis to make it practical.

Our approach to both problems is based on the observation that methods generally implement small and coherent pieces of functionality. Therefore, the runtime events issued during a method's execution are related to each other. We build upon this relation and define:

```
class A {
  public void callM(LinkedList<B> li) {
    Iterator<B> iter = li.iterator();
    while (iter.hasNext()) {
      B b = iter.next();
      b.m();
    }
  }
}

class B {
  public void m() {
    n();
    f.g();     // f is some field of B
  }

  private void n() { /* ... */ }
}
```

Figure 3. Method *callM()* uses an iterator to traverse a list.

Object	Method Calls (Role)
li	iterator()
iter	hasNext(), next()
instance of B	m()
(more instances of B)	m()

Table I
AN OBJECT COLLABORATION RESULTING FROM THE EXECUTION OF *callM()* FROM FIGURE 3.

Definition: Object collaboration
The sequence of method calls within the execution of a method and their receiver objects form an object collaboration.
More formally, let a method call be a pair (o, s) of the receiver object o and the called method's signature s. A collaboration is an ordered sequence

$$S = (o_1, s_1), \ldots, (o_n, s_n)$$

of calls issued within the execution of a method (o_{outer}, s_{outer}). The objects

$$O = \{o \mid \exists(o, s) \in S\}$$

are said to collaborate. Note that an individual object or method signature can appear multiple times in S.

Building object collaborations from method traces yields small sets of related objects and method calls to reason about. The main advantage is that one can analyze each object collaboration separately, without considering other collaborations in the method traces.

As an example, consider the Java code in Figure 3. An execution of *callM()* involves multiple objects: the collection *li*, the iterator *iter*, and a number of instances of *B* accessed via the variable *b*. The resulting object collaboration consists of the objects and method calls given in Table I. Although the order of calls is missing in this representation, the order is preserved internally for each collaboration.

To make it more applicable, we refine the definition of collaborations in two ways. First, each of the called methods can involve more objects and method calls. Recursively considering all method calls originating in a certain method can yield an infinite set. Therefore, we limit the depth of nested calls to a certain *nesting level*. The collaboration in Table I is build with nesting level one. For nesting level two, we would add method *n()* to the instances of *B* because *n()* is called during the execution of *m()*. Moreover, we would include the objects that *B*'s field *f* refers to and list the method *g()* for each of them. As the example illustrates, one can vary the scope of object collaborations by changing the nesting level.

Second, programmers often split the functionality of a method into smaller, mostly private methods in the same class. In that case, executing the outer method includes, seen from outside of the object, the execution of the inner methods. We consider this common programming style by *inlining* all calls of an object to its own methods when we build object collaborations. In the example of Figure 3, *m()*'s call to *n()* is considered to be part of *m()*'s execution. Hence, it is inlined and does not count as a nested call.

The next step of our analysis identifies frequent execution patterns in the set of object collaborations.

C. Pattern Extractor

To effectively learn a FSM that describes common method usages, we need runtime event sequences that are similar but expose different orders of calls. We say that such runtime event sequences belong to a *collaboration pattern* and derive these patterns from object collaborations.

In the context of a collaboration, a certain set of methods is called on each involved object. Comparing conceptually similar execution sequences (such as different uses of iterators), we observed that the involved objects often require the same sets of methods. We exploit this observation for extracting patterns:

Definition: Role
The role of an object in a collaboration is the subset of its methods that are called in the scope of the collaboration.
In other words, if an object $o \in O$, then

$$role(o) = \{s \mid \exists (o, s) \in S\}.$$

The order of calls is not considered in a role. A particular object can play different roles in different collaborations. Likewise, there can be multiple objects with the same role in one collaboration. In Table I, the right column shows the roles played by the objects of the collaboration.

Based on this definition of the role of an object, we apply a set of techniques that make the similarities between object collaborations explicit:

1) Generalizing types of objects to the most general type that fulfills their role;
2) Merging objects that play the same role, for instance, due to an iteration that calls the same methods on each object of a collection;
3) Filtering of irrelevant objects and method calls, assuming that related events relate to the same package.

In the following, we detail these three techniques.

Generalizing types: Our goal in generalizing the types of objects in a collaboration is to identify similarities between instances of classes with a common supertype (class or interface). The role of an object abstracts from the ordering of methods called on it, which helps in identifying similar object usages. We can identify even more similarities using the observation that the role of an object mostly contains fewer methods than the actual type of the object. Hence, we assign to each involved object a type that has all methods of its role but no more additional methods than necessary. In Figure 3, the dynamic type of variable *li* is *LinkedList* and that of *iter* is *ListItr*[1]. However, *li*'s role is $\{$*iterator()*$\}$ and *iter*'s role is $\{$*hasNext(), next()*$\}$. Thus, their supertypes *Iterable* and *Iterator*, which contain these methods, are sufficient in this collaboration.

In general, the best-fitting type for an object o with respect to a role r is determined by considering all supertypes of o's dynamic type that fulfills r. For each such supertype t, we calculate the ratio between the number of methods in r and the number of accessible methods in t:

$$fit(r, t) = \frac{|methods(r)|}{|methods(t)|}$$

Afterwards, we choose the type that maximizes fit, or in other words, that minimizes the number of methods that are not required in the collaboration. If a type provides exactly the role's methods, fit equals one, such as in Figure 3 for *li* and *Iterable*, which only contains the *iterator()* method.

To illustrate the effect of generalizing types, consider the code in Figure 4 and compare it to Figure 3. Comparing the method traces of executions of *addElements()* and *callM()* shows no obvious similarities apart from common method names. Focusing on each object's role and generalizing their types, though, reveals that both executions contain one *Iterable* object and one *Iterator* object, and that the same methods are called on these objects.

Merging objects: Iterations over collections are a particular problem for inferring specifications, because the number of elements in a collection can vary between executions. We deal with that problem by merging these elements into a

[1]*java.util.LinkedList.ListItr* is an inner class of *LinkedList* in Sun's Java standard library (version 1.6.0).

```
public class C {
  public int addElements(HashSet<Integer> s) {
    int sum = 0;
    for (Iterator<Integer> i = s.iterator();
         i.hasNext(); ) {
      sum += i.next();
    }
    return sum;
  }
}
```

Figure 4. Method *addElements()* iterates over a hash set.

single element. If multiple objects play exactly the same role in a collaboration, they are reduced to one artificial object that receives all method calls to the merged objects. For example, executing *callM()* from Figure 3 yields arbitrarily many instances of *B*, which all play the same role. We merge all these instances into one object, and as a result, executions of *callM()* with lists of different length appear to be the same.

Package-based filtering: Another obstacle to every specification inference technique is that events that are relevant for a particular behavioral pattern are interleaved with irrelevant ones. For instance, the *callM()* method contains calls to *B.m()* which are not relevant for inferring specifications of correct iterator usage. Our analysis handles that problem by assuming that related method calls deal with objects from the same package. Specifically, a call is considered to relate to a package *p* if:

- the callee is an instance of a class in *p*,
- the return value is an instance of a class in *p*, or
- one of the arguments is an instance of a class in *p*.

Package-based filtering can be run in two modes. One option for a user of our analysis is to specify a particular package of interest to only consider runtime events related to this package. Alternatively, the analysis automatically identifies potentially interesting packages by considering for each collaboration all those packages to which at least two method calls relate. For each such package, we produce a package-specific collaboration by removing all method calls that do not relate to the package. That is, one "real" collaboration can result in multiple filtered collaborations, where each filtered collaboration only contains the method calls that relate to a particular package.

Our approach for package-based filtering refines ideas from Weimer et al., where two method calls can only lead to a temporal specification if both methods are declared in the same package [5]. In addition, we also consider calls that are in a data-flow relation to an object of a particular package.

Generalizing types, merging objects, and filtering collaborations by package are techniques to highlight the similarities between different collaborations. After applying them, we map collaborations into patterns by comparing their sets of roles:

Definition: Collaboration pattern
Two collaborations belong to the same collaboration pattern if their objects play the same roles. That is, given two collaborations with object sets O_1 and O_2, they belong to the same pattern if and only if there exists a bijective map $m : O_1 \rightarrow O_2$ with

$$o_1 \mapsto o_2 \in m \Leftrightarrow role(o_1) = role(o_2).$$

Since roles abstract from the order of calls, mapping collaborations to patterns also suppresses the order. Instead, we only consider which methods are called on the individual objects. The two examples in Figures 3 and 4 lead to a common pattern if one filters events related to *java.util*. Both, executing *callM()* and executing *addElements()*, involve an *Iterable* object with role $\{$*iterator()*$\}$ and an *Iterator* object with role $\{$*hasNext(), next()*$\}$. That is, there is a one-to-one correspondence between the involved objects, and hence, we assign them to the same pattern.

Ranking: Deriving patterns of object collaborations drastically reduces the volume of data compared to the execution traces they are obtained from. However, for realistic applications, the number of patterns is still rather large and, naturally, not all of them contain characteristic behavior. We propose three criteria to rank patterns and focus on the more interesting ones.

First, we disapprove patterns whose collaborations involve many objects, since they lead to FSMs that are rather complex and too large to be useful to humans. A reasonable upper limit on the number of involved objects is in the range between 5 and 10.

Second, we consider the *dynamic frequency* of a pattern, that is, the number of times it was found in the method traces. Recall that we assume correct behavior to occur frequently. Inversely, this means that patterns with low dynamic frequency are less reliable for inferring specifications of correct behavior. Therefore, reliability is improved by focusing on patterns whose dynamic frequency is above a certain threshold.

Third, patterns that occur in multiple different methods are likely to be more reliable. We call the number of statically different call sites that lead to a pattern its *static frequency*. For example, the iterator usage pattern derived from Figures 3 and 4 has a static frequency of at least two, because it occurs in the two methods *callM()* and *addElements()*.

This concludes the description of a set of techniques to identify frequently occurring execution patterns in object collaborations. In the following, we explain how we infer specifications from these patterns.

D. *Specification Inference*

The final step of our analysis is to derive FSMs that model legal sequences of method calls on a set of related objects. We create a FSM for each collaboration pattern using the following mapping:

- *States.* For each method of an object that is called in a collaboration of the pattern, create a new state. If there are multiple objects on which the same method is called, create a state for each object-method pair.
- *Transitions.* For each pair of consecutive method calls *(m(), n())*, create a transition from the state representing *m()* to the state representing *n()*. Assign weight one to it. If such a transition exists already, increase its weight by one.

For example, Figure 1 is a FSM derived from methods that iterate over collections using iterators, such as *callM()* and *addElements()*. We found this FSM in several applications we analyzed.

A limitation of the way we construct FSMs is that each method is represented by only one state, even if multiple paths lead to a call of a particular method. An alternative are anonymous states and transitions that represent method calls [11], [8], [10], [6]. In such FSMs, the presumably permitted method call sequences correspond to the accepted language. Unfortunately, learning the smallest such FSM from a finite set of traces is NP-hard [12]. Numerous approximations have been proposed that generalize the accepted language, and hence, accept more call sequences than the given ones [13]. In contrast, our approach (also taken by others [9]) makes building FSMs straightforward and does not require approximations. Moreover, weighting the transitions by their frequency allows users to estimate the reliability of each transition.

III. Implementation

To evaluate our approach, we have implemented the analysis in the Scala programming language [14]. For instrumenting applications, we use aspect-oriented programming and the AspectJ compiler [15]. An aspect with two pointcuts adds instructions before each method call and after each method return. The corresponding advices pass the runtime information to a logging module, which writes the method call and return events into log files. For multi-threaded applications, we create a separate log file for each thread. Thus, the order of calls is preserved per thread, but not globally. The method traces are stored in plain text files. The largest file we analyzed is about 7.1 GB in size and contains more than 54 million runtime events.

We instrument only classes that belong to the analyzed application and omit libraries and frameworks. In particular, we do not instrument the Java standard library. As a result, the generated FSMs contain API usage patterns, which otherwise would be interleaved with events from inside libraries and frameworks.

The analysis itself has an input reading module, which transforms log files of method traces into a stream of object collaborations. For assigning collaborations to patterns, we need to know the static structure of the application, for example, to retrieve the supertypes of a class. This part is implemented using Java's reflection facilities, and hence, we need access to the bytecode (but not the source code) of the application. The main output of our tool is a set of DOT files [16], each containing one FSM, which we generate using the *dk.brics.automaton* library [17].

IV. Results

This section describes our experiences from applying the proposed approach to several real-world applications. At first, we give an overview of the applications used in the experiments. Second, we analyze the runtime performance and scalability of our implementation and show that it runs significantly faster than comparable approaches. Third, we evaluate the quality of the generated specifications. For this aspect, we analyze different applications that use the same library and compare their results. Finally, we demonstrate the ability of our approach to produce API usage documentation. We show that existing textual descriptions of typical and correct method call sequences from a commonly used Java reference [18] can be recovered by our analysis.

A. *Overview of Experiments*

Table II shows the ten applications we used for most of the experiments. They are part of the DaCapo benchmark suite, which has the benefit of ensuring a controlled and reproducible execution of all applications [19]. For each application, the number of loaded classes (taken from [19]) and the number of analyzed runtime events is given, where each event is either a method call or a method return. Overall, we analyzed more than 240 million events. Furthermore, Table II provides the number of identified object collaborations and the number of patterns or FSMs derived from them.[2] The results show that our approach derives a manageable number of specifications from a large volume of method traces.

Note that the results in Table II refer to a run of the analysis for object collaborations in all packages. Restricting the analysis to a particular package reduces the number of FSMs as we show in Section IV-C.

All experiments are done on a 3.16 GHz Intel Core 2 Duo machine with 4 GB memory. We use the Java Hotspot Server virtual machine version 1.6.0 running on Debian/GNU Linux. Execution times are measured with the Linux *time* command.

[2]The number of collaborations only includes those with at least one method call. That is, a call to a method that returns without making any other call does not count as a collaboration.

Application	Classes Loaded	Events	Collaborations	FSMs
antlr	126	18,718,204	1,243,780	262
chart	219	28,875,810	3,862,532	67
eclipse	795	498,328	39,542	491
fop	231	6,813,674	1,045,194	391
hsqldb	131	18,617,950	829,802	244
jython	251	54,204,213	2,742,132	365
luindex	128	38,719,540	1,457,636	46
lusearch	118	21,080,277	1,232,174	45
pmd	325	295,750	11,632	90
xalan	244	56,646,196	8,951,174	581
Overall	2,568	244,469,942	21,415,598	2,582

Table II
OVERVIEW OF THE APPLICATIONS USED IN OUR EXPERIMENTS.

B. Scalability

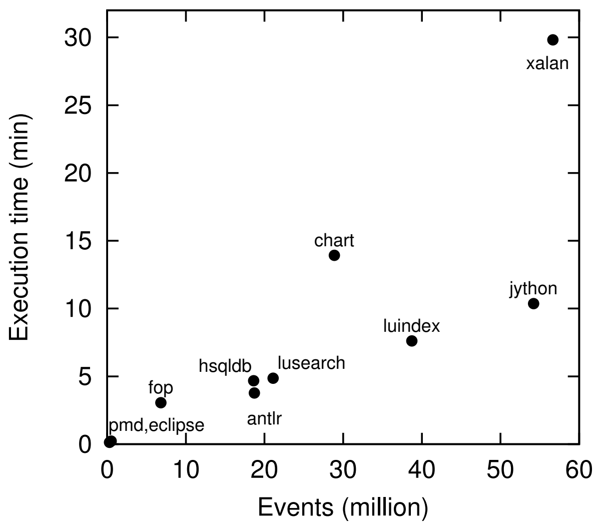
Our specification inference algorithm scales linearly with the number of runtime events given to it as input. As a result, method traces of real-world applications can be analyzed in a few seconds or minutes.

The main reason for the scalability of our approach is that each collaboration can be analyzed locally, without knowledge about prior or future collaborations. Afterwards, each collaboration is assigned to a pattern based on its set of roles. If no matching pattern exists, a new pattern is created. As described in Section II-D, each pattern corresponds to a FSM. Thus, the only global data structure of the analysis is the set of resulting FSMs.

Figure 5 depicts the execution times of the analysis for the applications in Table II. The execution times range between 7.8 seconds (*pmd*) and almost 30 minutes (*xalan*). The upper graph shows execution time against number of runtime events. Two executions (*chart* and *xalan*) take some more time per event than the others. We analyzed the reason for the two exceptions and found that the method traces of *chart* and *xalan* yield exceptionally many object collaborations. For illustration, consider the lower graph in Figure 5, which shows execution time against number of collaborations. Here, all executions perform approximately linearly.

Figure 5 shows the execution times required to extract specifications for all packages. The analysis runs even faster if a particular package is selected. For example, if the focus is on package *java.util*, the execution time of the analysis ranges between 6.3 seconds and 21 minutes.

In summary, the measurements confirm our reasoning that the runtime of our analysis is roughly linearly dependent on the size of its input. The analysis runs significantly faster than existing approaches [20], [10], even though our implementation is not particularly optimized for speed. As



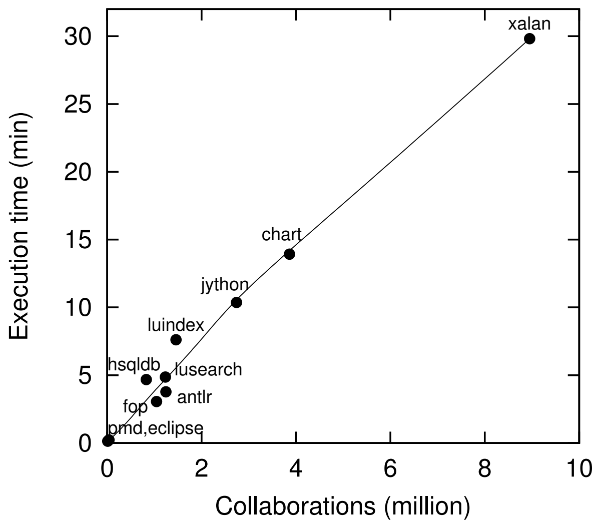


Figure 5. Execution time of the analysis against the number of analyzed events (above) and against the number of identified collaborations (below).

a result, not more than a few minutes are required for analyzing traces from real applications.

C. Quality of Generated Specifications

We use raw runtime data as input and do not make any approximation that could introduce false behavior. Therefore, the generated FSMs must be accurate in terms of only describing possible method call sequences. However, our technique entails the risk to highlight incidental call sequences that, although occurring in the method traces, are not characteristic in general.

Therefore, the crucial question for evaluating the quality of our results is how many of the produced specifications describe typical object usages. We measure the proportion of typical FSMs by comparing the results from different

Application	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%

Table III
PROPORTION OF CONFIRMED AND UNCONFIRMED SPECIFICATIONS FOR THE *java.util* PACKAGE.

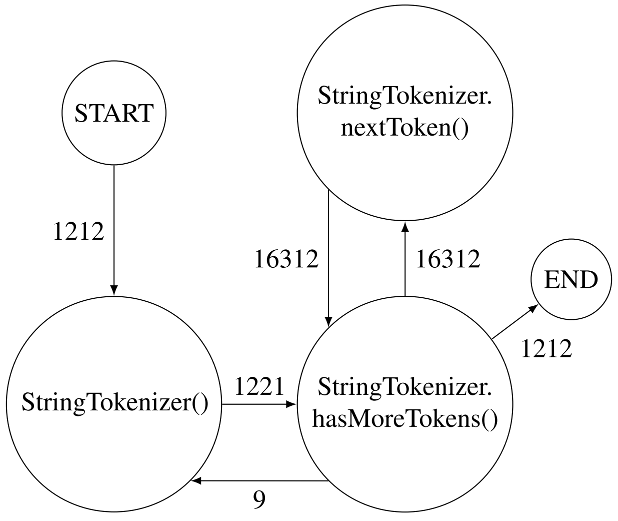


Figure 6. Specification inferred from the *jython* project.

applications that use the same API. We consider a specification to be typical if we find it in the results of at least two applications. More specifically, each generated FSM belongs to one of the following categories:

- *Confirmed by identity*, if the same FSM is also derived from another application;
- *Confirmed by inclusion*, if the FSM is included in some FSM derived from another application. A FSM M_1 is included in M_2 if all states of M_1 appear in M_2 and if the transitions of M_1 are a subset of the transitions of the corresponding states of M_2;
- *Unconfirmed*, if none of the above applies.

To measure the percentage of confirmed FSMs, we generate specifications for the utility package *java.util* and compare the results from ten applications. Table III shows the categorization into confirmed and unconfirmed specifications. Overall, our tool generates 247 specifications. More than a third (35%) of these specifications 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 patterns and not incidental call sequences. Figures 1 and 6 are two examples of the specifications generated for *java.util*, originating from the *pmd* and *jython* projects, respectively.

D. Recovery of Documentation

Specifications can be used as documentation. In contrast to a natural language, formal specifications are less ambiguous and can be processed automatically. To evaluate whether our inference technique produces FSMs that document correct API usage, we tried to recover existing documentation given in textual form (inspired by Xie [21]).

A widely used standard reference provides the following documentation on using Java's *ZipFile* class [18]:

> A *ZipFile* can be created by specifying the ZIP file to be read either as *String* filename or as a *File* object. ... Once a *ZipFile* is created, the *getEntry()* method returns a *ZipEntry* object for a named entry ... To read the contents of a specific *ZipEntry* within the ZIP file, pass the *ZipEntry* to *getInputStream()*; this returns an *InputStream* object from which you can read the entry's contents.

To recover this documentation artifact, we instrumented *jEdit* [22], because it has several plugins that read and write ZIP files. An analysis of *jEdit*'s traces, restricted to the *java.util.zip* package, produces four FSMs, including Figure 7. The specification in Figure 7 corresponds to the documentation given in [18].

During our experiments, we found more FSMs that bear a resemblance to existing textual documentation. This shows that our approach is able to generate documentation artifacts. We conclude that, given representative applications using an API, our techniques can be applied to generate documentation of how to use the API.

V. DISCUSSION

The applications used in the experiments cover a wide range of domains from program analysis to databases. Therefore, we consider our results concerning scalability (Section IV-B) and quality of the inferred specifications (Section IV-C) to be generalizable to other applications. The preliminary results on documentation recovery (Section IV-D) are promising and should be validated in a separate case study involving several APIs and applications.

Although our experiments focus on Java programs, we believe that the techniques presented in this paper can be

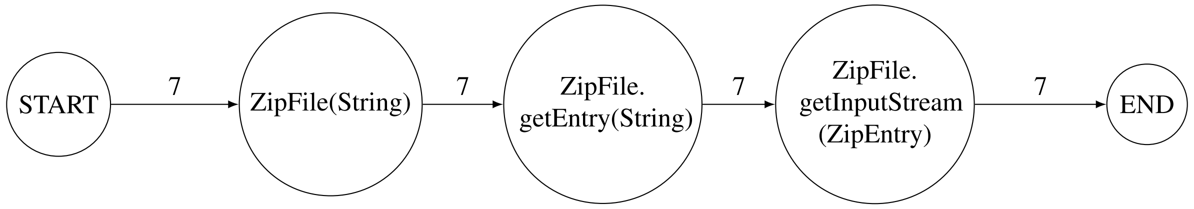


Figure 7. Documentation of *ZipFile* that we recovered from running *jEdit*.

adapted to other programming languages. Our method trace format fits any statically typed object-oriented language, and the algorithm for identifying object collaborations is not tied to particular Java features. In our implementation, those parts of the analysis that use the static structure of the analyzed application are separated and can easily be adapted to support other languages.

Given our results, we consider object collaborations to be a useful abstraction for specification inference. However, some specifications that developers could be interested in may be scattered over different, unrelated methods. For example, patterns of using a global data structure may not occur in any object collaboration. Also, our definition of collaborations considers only control flow and ignores data flow relations between runtime events. Nevertheless, our experiments suggest that object collaborations are a useful abstraction. In particular, we have shown their benefits for scalability since we can analyze them locally. Since methods are the main abstraction to structure functionality and behavior in object-oriented programming, it seems a reasonable conclusion to exploit this structure for finding behavioral patterns.

A possible development of our ideas is to use them as input for static and dynamic verification techniques. There are several frameworks for monitoring runtime events, such as method calls, which trigger additional functionality when particular patterns of events occur [23], [24], [25]. These frameworks support patterns that involve multiple objects. Hence, the FSMs generated by our analysis can be combined with these frameworks to search for violations of the specifications. On the same lines, static analyses, such as type state systems [26], [27], are possible applications of our results.

VI. Related Work

A. *Specification Mining*

Ernst et al. were among the first that derived specifications from runtime data [28], [29]. Their work extracts invariants, which possibly involve multiple variables, from program traces.

There have been several proposals for deriving specifications of legal calls sequences from runtime data. Ammons et al. apply a probabilistic FSM learner to method traces of C programs to derive specifications of correct API usage [8]. Gabel et al. advance the idea of specification inference as language learning problem and present Javert, a tool to extract FSMs that involve multiple objects [10]. Their analysis reduces the complexity of the problem by focusing on two predefined micro-patterns. Perracotta is a specification mining tool that focuses on scalability and possibly imperfect traces [20]. Similar to [10], the approach is to search method traces for instances of certain templates, such as two events that may only occur in strictly alternating order. Rather than searching instances of pre-defined templates, our approach can identify arbitrary usage patterns, and still ensures scalability as shown in Section IV-B. Our work also differs from the above in the kind of FSMs that result. Instead of using an approximative language learning algorithm, we map methods to states and obtain FSMs that precisely describe the observed behavior.

Ghezzi et al. [30] infer specifications of container-like classes as FSMs combined with graph transformation rules. This combination describes the visible state of the specified classes more fine-grained than plain FSMs. In contrast to our approach, though, only single classes are specified. Lo and Khoo [31] propose a framework for dynamically mining API specifications. It consists of modules for trace-filtering, clustering, learning, and automata-merging. Similar to [8], a probabilistic FSM learner is used. The main focus of the work is on inferring precise specifications instead of scalability. Reiss et al. discuss space-reducing encodings of runtime data and, as a by-product, propose an algorithm for inferring FSMs from method call sequences [32]. Salah et al. infer different usage scenarios for a single class by grouping similar method invocation sequences on instances of this class into canonical sets [33].

Besides these dynamic approaches, different static analyses for the automated generation of programming rules or specifications have been proposed. Whaley et al. infer FSMs describing all call sequences to an instance of a class that do not lead to an exception [9]. The work is based on the assumption that programmers use some field of a class to track its internal state. The analysis identifies sets of related methods by considering those subsets of a class' methods that access a common field. Alur et al. present a generalization of that approach [11]. Their analysis searches the most general temporal interface of a class that does not lead to an unsafe valuation of the class' fields. The above analyses are conservative so that the produced specifications permit more call sequences than those that are actually legal. On the contrary, specifications inferred with our approach are rather restrictive and include only the observed behavior.

Wasylkowski et al. present a static analyzer that finds temporal properties, such as that *n()* can be called after *m()* [6]. As an intermediate step, their analysis constructs FSMs that focus on the use of one particular object in a method. Acharya et al. derive partial orders of API method calls from C source code [34]. Liu et al. check the likelihood of API usage patterns with a model checker that counts the number of validations and violations for each pattern candidate [35]. PR-Miner is a tool to extract implicit programming rules from C software [4]. It analyses variables and functions that are used together in a function and identifies frequent combinations of them using frequent itemset mining.

The discovery of algebraic specifications for Java container classes is described by Henkel et al. [7]. An algebraic specification consists of axioms that describe equalities resulting from different sequences of method calls. Their approach generates such axioms automatically and tests them for validity using generated unit tests.

B. Bug Finding and Verification

Among the various applications of generated specifications, identifying potential bugs is one of the most investigated. Several of the above mentioned works contain static [4], [6] or dynamic [9] techniques to detect violations of the inferred rules. A static analysis by Engler et al. finds frequently occurring instances of rule-templates and spots potential errors as violations of the rules [3]. Weimer et al. present an analysis that identifies potential bugs in error handling code by searching for unusual pairs of method calls in exceptional control flow paths [5].

The specifications produced by our analysis can be given as input to existing verification techniques. Both dynamic checking, which can emit warnings when methods other than specified are called, and static checking, such as type state analysis [26], [27], are possible.

C. Other Related Work

Our notion of roles and collaborations is inspired by work in conceptual and object-oriented modeling [36] and program understanding [37]. In both, a role describes the responsibilities of an object in a certain context. In our model, this context is an object collaboration that happens within the execution of a method. Dynamic object process graphs are another way to extract small sets of related events from dynamic traces [38]. They can also be used for inferring specifications [39]. An object process graph is a view on a control flow path with focus on a single object. Its vertices are locations in a program; edges correspond to control flow between these locations. In contrast, object collaborations focus on the execution of a single method and can involve multiple objects.

Livshits et al. propose a technique for identifying violations of application-specific programming rules with the help of software revision repositories [40]. Similar to us, they search for sets of related methods. Livshits et al. assume that methods that occur in the same check-in relate to each other and apply a data mining algorithm to spot frequent method sets. A system for detecting code injection attacks by Fetzer et al. includes a component for learning typical system call sequences from method traces based on data-flow relations between calls [41]. Xie and Notkin explore the synergistic effects of specification inference and test generation [42]. They propose an iterative process that derives specifications from test executions, which in turn are used to enhance an existing test suite.

Cook et al. compare different methods for deriving FSMs that describe software development processes [43]. Despite the differing application domain, their work is related, since they also infer temporal rules from sequences of observed events.

VII. Conclusions

This paper addresses the question how to infer specifications of legal method call sequences from traces of runtime events in a practical manner. Formal specifications can be used for various software engineering activities ranging from finding errors to documenting software. Our analysis automates the generation of specifications, and hence, is a step toward making specification-based activities accessible without the need to write them manually.

This paper improves existing approaches in several ways: First, we show that interesting specifications can be inferred in reasonable time. Our experiments suggest the analysis to scale linearly with the size of method traces in most cases, so that it can handle realistic applications. Second, we show that regarding the roles that objects play in a certain context helps in identifying similar execution sequences, which is required for inferring temporal specifications. Third, our work differs from most existing approaches by providing specifications for multiple objects rather than single objects or classes. Consequently, our results contain more complex specifications, which may be hard to deduce manually. A potential shortcoming of our approach is that we may miss programming rules that exceed the scope of object collaborations. Also, our analysis cannot guarantee the completeness of the inferred specifications.

We envision different applications of our ideas in future work. One application is to use the generated specifications as input for runtime verification or static analysis, to identify uncommon behavior. Such an approach can spot potential errors, unusual programming patterns, or disobedience to project-internal programming rules. Furthermore, our preliminary results on recovering existing documentation are promising, and we plan to evaluate the ability of our algorithm to infer documentation in a larger case study.

Acknowledgments

We would like to thank the anonymous reviewers, as well as Zoltán Majó and Marco Zimmerling for providing valuable feedback on how to improve this paper. We thank Stephanie Balzer, Nicholas Matsakis, and Friedrich Steimann for discussions of this work. Also thanks to Andreas Møller for his help on using the *dk.brics.automaton* library.

References

[1] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2000, pp. 1–16.

[2] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *Symposium on Principles of Programming Languages (POPL)*, 2002, pp. 1–3.

[3] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Symposium on Operating Systems Principles*, 2001, pp. 57–72.

[4] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2005, pp. 306–315.

[5] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005, pp. 461–476.

[6] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2007, pp. 35–44.

[7] J. Henkel, C. Reichenbach, and A. Diwan, "Discovering documentation for Java container classes," *IEEE Transactions on Software Engineering*, vol. 33, no. 8, pp. 526–543, 2007.

[8] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Symposium on Principles of Programming Languages (POPL)*, 2002, pp. 4–16.

[9] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *Symposium on Software Testing and Analysis (ISSTA)*, 2002, pp. 218–228.

[10] M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in *Symposium on Foundations of Software Engineering (FSE)*, 2008, pp. 339–349.

[11] R. Alur, P. Cerný, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for Java classes," in *Symposium on Principles of Programming Languages (POPL)*, 2005, pp. 98–109.

[12] E. M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, no. 3, pp. 302–320, 1978.

[13] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behaviour," *IEEE Transactions on Computers*, vol. 21, pp. 592–597, 1972.

[14] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala, A comprehensive step-by-step guide*. Artima, 2008.

[15] "The AspectJ project," http://www.eclipse.org/aspectj/.

[16] "Graphviz - graph visualization software," http://www.graphviz.org/.

[17] A. Møller, "Finite-state automata and regular expressions for Java," http://www.brics.dk/automaton/.

[18] D. Flanagan, *Java in a nutshell*. O'Reilly, 2005.

[19] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006, pp. 169–190.

[20] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *International Conference on Software Engineering (ICSE)*, 2006, pp. 282–291.

[21] T. Xie, "Software component protocol inference," University of Washington, Tech. Rep., June 2003. [Online]. Available: http://www.csc.ncsu.edu/faculty/xie/publications/generals-tao.pdf

[22] "jEdit - programmer's text editor," http://www.jedit.org/.

[23] M. C. Martin, V. B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: A program query language," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005, pp. 365–383.

[24] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005, pp. 345–364.

[25] F. Chen and G. Rosu, "MOP: An efficient and generic runtime verification framework," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007, pp. 569–588.

[26] R. E. Strom and S. Yemini, “Typestate: A programming language concept for enhancing software reliability,” *IEEE Transactions on Software Engineering*, vol. 12, no. 1, pp. 157–171, 1986.

[27] K. Bierhoff and J. Aldrich, “Modular typestate checking of aliased objects,” in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007, pp. 301–320.

[28] M. D. Ernst, “Dynamically discovering likely program invariants,” Ph.D. dissertation, University of Washington, 2000.

[29] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 213–224, 2001.

[30] C. Ghezzi, A. Mocci, and M. Monga, “Synthesizing intensional behavior models by graph transformation,” in *International Conference on Software Engineering (ICSE)*, 2009, pp. 430–440.

[31] D. Lo and S.-C. Khoo, “Smartic: towards building an accurate, robust and scalable specification miner,” in *Symposium on Foundations of Software Engineering (FSE)*, 2006, pp. 265–275.

[32] S. P. Reiss and M. Renieris, “Encoding program executions,” in *International Conference on Software Engineering (ICSE)*, 2001, pp. 221–230.

[33] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos, “Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences,” in *Conference on Software Maintenance (ICSM)*, 2005, pp. 155–164.

[34] M. Acharya, T. Xie, J. Pei, and J. Xu, “Mining API patterns as partial orders from source code: From usage scenarios to specifications,” in *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2007, pp. 25–34.

[35] C. Liu, E. Ye, and D. J. Richardson, “Software library usage pattern extraction using a software model checker,” in *Automated Software Engineering (ASE)*, 2006, pp. 301–304.

[36] F. Steimann, “On the representation of roles in object-oriented and conceptual modelling,” *Data & Knowledge Engineering*, vol. 35, no. 1, pp. 83–106, 2000.

[37] T. Richner and S. Ducasse, “Using dynamic information for the iterative recovery of collaborations and roles,” in *International Conference on Software Maintenance (ICSM)*, 2002, pp. 34–43.

[38] J. Quante and R. Koschke, “Dynamic object process graphs,” *Journal of Systems and Software*, vol. 81, no. 4, pp. 481–501, 2008.

[39] ——, “Dynamic protocol recovery,” in *Working Conference on Reverse Engineering (WCRE)*, 2007, pp. 219–228.

[40] V. B. Livshits and T. Zimmermann, “DynaMine: Finding common error patterns by mining software revision histories,” in *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2005, pp. 296–305.

[41] C. Fetzer and M. Süßkraut, “Switchblade: Enforcing dynamic personalized system call models,” in *EuroSys*, 2008, pp. 273–286.

[42] T. Xie and D. Notkin, “Mutually enhancing test generation and specification inference,” in *Formal Approaches to Software Testing*, 2003, pp. 60–69.

[43] J. E. Cook and A. L. Wolf, “Discovering models of software processes from event-based data,” *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 215–249, 1998.