A Framework for the Evaluation of Specification Miners Based on Finite State Machines

Michael Pradel
Computer Science Department
ETH Zurich

Philipp Bichsel
Computer Science Department
ETH Zurich

Thomas R. Gross
Computer Science Department
ETH Zurich

***Abstract*—Software maintenance tasks, such as testing and program understanding, can benefit from formal specifications that describe how a program should use an API. Recently, there has been increasing interest in *specification miners* that automatically extract finite state specifications of method ordering constraints from existing software. However, comparing different mining approaches is difficult, because no common ground to evaluate the effectiveness of specification miners has been established yet. We present a framework for evaluating to which extent specification miners find valid finite state descriptions of API usage constraints. The framework helps in creating reference specifications and includes metrics to compare mined specifications to the reference specifications. The metrics are tailored for evaluating specification miners and account for imprecision and incompleteness in mined specifications. We use the framework to compare the effectiveness of three mining approaches and to show their respective benefits.**

***Index Terms*—Requirements/Specifications, Mining, Metrics/ Measurement**

I. Introduction

Typical application programming interfaces (APIs) provide a large set of methods to programmers. However, not every method can be called at any point during the execution of a program. Instead, programmers must be aware of method ordering constraints that have to be respected to use an API correctly. For instance, to use an input stream of the Java I/O API, a programmer must instantiate a stream class, read from the stream, and eventually close it.

Unfortunately, method ordering constraints of APIs are usually not explicitly specified. To address this problem, various approaches towards *mining specifications* have been proposed [1]–[21]. Specification miners analyze existing client programs that use an API, the library or framework that provides the API, or other available data of a software project, such as a version history. From this input, miners extract specifications of commonly observed behavior. Mined specifications can be used for different purposes, such as finding programming errors [5], [7], [13], [18] and program understanding [3], [12].

In this paper, we focus on specification miners that produce finite state machines (FSMs) describing method ordering constraints. For example, Figure 1a shows an FSM describing the usage of the *java.util.Formatter* class. In the FSMs given in this paper, states represent the state of one or more objects; transitions indicate method calls and how these calls change the state of the involved objects.

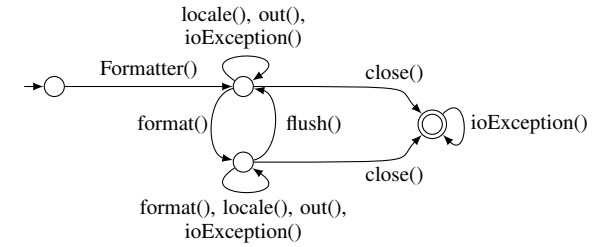
The existence of specification miners that automatically infer such FSMs leads to the question how to assess the effectiveness of a mining approach. Currently, there is no common ground on which to assess the quality of mined specifications. As a result, it is difficult to evaluate specification miners in a reproducible way and to compare them with each other.

The current state of the art to evaluate specification miners is to manually classify mined specifications [11], [12], [14], [16], [19] or to manually compare them to a reference [1], [2], [5], [20]. A manual classification and comparison of specifications risks to be subjective and is not reproducible. Manual assessment is also limited to two outcomes: valid and invalid. Another evaluation technique is to show the usefulness of mined specifications for some task, such as finding bugs [5], [7], [13], [18]. Unfortunately, this approach does not allow for assessing the effectiveness of the mining approach itself.

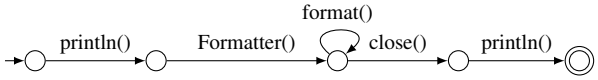
To address the problems of current evaluation techniques, we present a novel framework for the evaluation of specification miners. The evaluation framework allows for assessing the quality of mined specifications in a reproducible way. As a result, one can compare different mining approaches in an objective manner. The framework consists of two parts: First, we present a lightweight specification mechanism that helps to formalize knowledge of API method ordering constraints given, for example, through informal API documentation. Based on these lightweight specifications, our framework generates FSMs that serve as a reference, or "golden standard", to evaluate mined specifications. Second, we present metrics for computing precision and recall [22] of a mined specification with respect to a reference specification. Intuitively, precision indicates how exact a mined specification is, while recall indicates how complete it is. Both metrics range between 0% and 100%, where 100% is the optimal result.

The most important property of our metrics is to account for incompleteness and imprecision in mined specifications. As a result, the metrics yield meaningful precision and recall values even if the FSMs contain additional method calls or miss important method calls, which is typically true for mined specifications. Previously proposed techniques for comparing FSMs are not robust enough against partial incorrectness of mined specifications [23] or compare FSMs on a structural level without considering the accepted language [24].

For example, consider the mined specification in Figure 1b. Compared to the reference specification in Figure 1a,



(a) Reference specification.



(b) Incomplete and imprecise mined specification.

Figure 1: Correct and complete specification of the *Formatter* class and a mined specification.

Figure 1b contains some valid parts, but includes several additional method calls and is missing large parts of the reference specification. To assess the quality of the mined specification, one can generate the reference in Figure 1a using our framework and compare both FSMs by computing precision and recall.

We use our evaluation framework to compare the results of three dynamic mining approaches. They differ in their way to extract subsequences of related method calls from large method traces. The first approach [21] is *method-centric* and extracts subsequences of calls based on the assumption that calls issued within the execution of the same method are related to each other. The second approach, which we devised to improve the first, is *object-centric* and extracts subsequences of calls to a particular API object (an instance of an API class) and other, related objects. The third approach is a simpler version of the second approach, which we call *single-object*. Here, each subsequence includes calls to a single API object, without considering any calls to other objects. Thus, the single-object approach can only find specifications involving single classes. We compare the results of these miners to 32 reference specifications of classes from the Java standard library, which we created using our framework. The results show the single-object miner to be the most successful in mining this kind of specifications. The miner achieves an average precision of 99% and a recall up to 45%. One reason for the high precision is that the single-object miner cannot consider calls to irrelevant classes, because each FSM it mines describes a single class.

In summary, the main contributions of this paper are:

1) A lightweight specification formalism to formally encode informal knowledge about method ordering constraints and an algorithm to translate these lightweight specifications into FSMs.
2) Metrics to compute precision and recall of FSMs with respect to a reference. The metrics are tailored for evaluating specification miners and account for imprecision and incompleteness in mined specifications.
3) An empirical evaluation of three mining approaches using our evaluation framework, which shows the benefits of the different approaches.

In the following section, we present the evaluation framework itself. Section III describes three mining approaches, which we compare in Section IV using our framework. Section V relates our approach to previous work and Section VI concludes the paper.

II. Evaluation Framework

Our framework evaluates mined specifications by comparing them to reference specifications. We define *reference specification* (or short, reference) as an FSM providing a precise and complete description of all method ordering constraints relevant for an API class or a group of related API classes. For example, Figure 1a shows the reference specification for the *Formatter* class. A fundamental question is whether specification miners should find complete specifications, or rather focus on smaller FSMs that omit rarely used methods and describe typical usages. Our framework compares mined specifications with complete specifications and assumes that miners strive for completeness. Complete specifications are more valuable for testing and verification and can be used as a basis for extracting smaller FSMs.

The evaluation framework consists of two parts: The first part addresses the problem of obtaining reference specifications by providing a semi-automatic approach to formalize existing, informal knowledge about method ordering constraints into FSMs. The second part addresses the problem of measuring how similar a mined specification is to a given reference. We present metrics to compute precision and recall. The metrics account for imprecision and incompleteness in mined specifications.

A. Building Reference Specifications

Method ordering constraints of real-world APIs are complex and involve many methods. As a result, manually formalizing them as FSMs is time-consuming and error-prone. To facilitate this process, we present a lightweight, yet concise and precise representation of ordering constraints between methods, which we call *Method Constraint Groups (MCGs)*. Our framework requires a human to encode method ordering constraints with MCGs, and then, automatically translates them into FSMs. The starting point for creating MCGs can be any commonly accepted source of knowledge describing the usage of an API, such as documentation or reference books.

MCGs organize all methods that are relevant for an API usage into groups based on their ordering constraints. The methods of a group do not impose any ordering constraints on each other, that is, they can be called in any order. Moreover, all methods of a group share the same constraints with respect to the methods in other groups. A method may be part of more than one group, which can be necessary to express different contexts in which a method may be used.

A MCG g consists of:

- a group identifier

ID	Methods	Enables	Disables	Final
0	-	1	-	No
1	Formatter()	2,3,5,6	1	No
2	Formatter.locale(), Formatter.out()	-	-	No
3	Formatter.format()	4	-	No
4	Formatter.flush()	-	4	No
5	Formatter.ioException()	-	-	No
6	Formatter.close()	7	2,3,4,5,6	Yes
7	Formatter.ioException()	-	-	Yes

Table I: Method Constraint Groups that describe ordering constraints between methods of *java.util.Formatter*.

Algorithm 1 Translate method constraint groups into an FSM

Input: Method constraint groups $G = \{g_i \mid 0 < i < n\}$
Output: FSM F representing the constraints of G

/* worklist of (state, group) pairs: */
1: $W \leftarrow W \cup \{(initialState(F), g_i) \mid g_i \in enables(g_0)\}$
/* map from sets of enabled groups to states: */
2: $S \leftarrow \{enables(g_0) \mapsto initialState(F)\}$
3: **while** $W \neq \emptyset$ **do**
4: $(s, g) \leftarrow$ take from W
/* groups enabled at destination state: */
5: $E \leftarrow (S^{-1}(s) \cup enables(g)) \setminus disables(g)$
6: **if** $E \in domain(S)$ **then**
7: $d \leftarrow S(E)$
8: **else**
9: $d \leftarrow$ create a new state
10: $S \leftarrow S \cup \{E \mapsto d\}$
11: $W \leftarrow W \cup \{(d, g_{enabled}) \mid g_{enabled} \in E\}$
12: **for all** $m \in methods(g)$ **do**
13: Add transition $s \xrightarrow{m} d$ to F
14: $isFinal(d) \leftarrow isFinal(g)$

- a set of methods
- *enables*, a set of identifiers of groups whose methods are enabled by calling a method from g
- *disables*, a set of identifiers of groups whose methods are disabled by calling a method from g
- *final*, a Boolean value indicating whether calling methods from g can terminate the API usage

Table I shows MCGs that describe the ordering constraints of methods from *Formatter*. As a convention, each set of MCGs has an empty group 0 that enables all groups not requiring a preceding call. As in the example in Table I, group 0 is often used to enable constructor calls. Instantiating a *Formatter* (group 1) enables most other groups and, at the same time, disables the constructor because calling the constructor again should not be part of the specification. Calling *close()* (group 5) disables most groups, since calling methods such as *format()* after *close()* leads to an exception.

While MCGs provide a precise and concise notation to encode ordering constraints between API methods, they cannot be easily compared to mined specifications given as FSMs. Therefore, we devised Algorithm 1, which translates MCGs into FSMs. Each state of the FSM build by Algorithm 1 represents a group of enabled methods, that is, methods that may be called in this state. The map S keeps track of the methods enabled in each state. If a transition leads to a new set of enabled methods, a new state is created (line 9). The algorithm maintains a worklist W of (state, group) pairs; for each pair (s, g) in W, the algorithm still must consider the methods of g as outgoing transitions of s. The worklist W is empty if and only if all call sequences that are permitted according to the MCGs are part of the language accepted by the FSM. The algorithm is guaranteed to terminate because there is a finite number of groups and methods.

Figure 1a is the result of applying the translation algorithm to the MCGs in Table I. We have chosen a rather simple example to illustrate the translation. As we show in Section IV-A, FSMs describing all possible call sequences of an API usage can be very large. For example, the complete specification for *LinkedList* and its iterators *ListIterator* and *Iterator* consists of 41 states and 465 transitions. In contrast, the corresponding MCGs contain 16 groups, where one group summarizes 35 methods that impose exactly the same constraints on other methods and no constraints on each other.

Another table-based description of FSMs are state transition tables, which have one row for each state and one column for each possible input (here: method). The large number of states and methods required for specifications of real-world APIs leads to large state transition tables, making them impractical for formalizing method ordering constraints.

By formalizing informal knowledge into MCGs and applying Algorithm 1, one can obtain reference FSMs that formally encode method ordering constraints. The advantage of using MCGs, instead of directly formalizing into FSMs, is that MCGs reduce the complexity of specifying large API classes by grouping methods according to their constraints.

B. Comparing Mined Specifications to Reference Specifications

Based on a set of reference specifications, we can evaluate to what extent mined specifications match the reference. For this purpose, we require a measure of similarity between a mined specification M and a reference specification R. In the following, we present two novel metrics to compute precision and recall of M with respect to R. Precision indicates what proportion of M matches R, that is, how exact M is, while recall indicates what proportion of R is present in M, that is, how complete M is.

The main challenge is to compute precision and recall in a way that accounts for noise and incompleteness in mined specifications. For example, consider the reference specification in Figure 1a and the three mined specifications in Figures 2a, 2b, and 1b. Each of the mined specifications partially matches the reference and our metrics should reflect these similarities.

To account for imprecision and incompleteness, our metrics use the k-tails algorithm [25]. This algorithm was originally proposed to synthesize FSMs from samples of their behavior. The k-tail of a state are all method call sequences of length k accepted in this state. For example, the 2-tail of the initial

state of Figure 2a contains four call sequences: *Formatter-locale*, *Formatter-out*, *Formatter-format*, and *Formatter-close*. The algorithm merges two states if they have the same k-tail. The parameter k controls how long the common tails of two states must be to get merged. We use a variant of the algorithm in which two states s_1 and s_2 are merged if the k-tail of s_1 is included in the k-tail of s_2. Variations of the k-tails algorithm are widely used to mine specifications [1], [8], [16], [26] and in other applications that synthesize FSMs [27], [28]. To the best of our knowledge, we are the first to apply the k-tails algorithm for comparing FSMs.

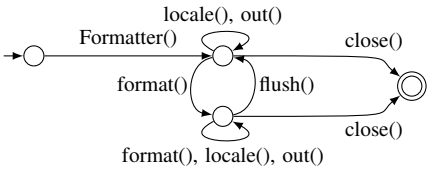
The following paragraph describes our algorithms for computing recall and precision of an FSM M with respect to an FSM R. Both algorithms are similar, and therefore, presented together. To compute precision (recall), we proceed as follows:

1) Compute the deterministic and minimal union automaton U of M and R and mark all transitions in U that come from M (from R).
2) Compute U' by applying the k-tails algorithm to U while propagating which transitions are marked. Make U' deterministic and minimize it.
3) Let m be the number of marked transitions in U'.
4) Compute the intersection of U' and R (intersection of U' and M) and let m_t be the number of marked transitions in U' taken while computing the intersection. The intersection (or product automaton) accepts all sentences accepted by both U' and R (U' and M). That is, when building the intersection all transitions required by both FSMs are taken. Transitions that are taken while computing the intersection but that are not marked correspond to unnecessary (missing) transitions in M.
5) The precision (recall) of M with respect to R is $\frac{m_t}{m}$.

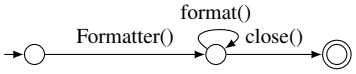
To propagate which transitions are marked, we adapt the k-tails algorithm and classical algorithms to minimize FSMs and to make FSMs deterministic [29] in a straightforward way: If a new transition t is created from transitions $t_1, \ldots, t_n$, then t is marked if and only if at least one of $t_1, \ldots, t_n$ is marked.

Figure 3 illustrates computing the recall of Figure 1b with respect to the reference in Figure 1a. First, we compute the union of both FSMs, as shown in Figure 3a. Transitions coming from the reference FSM are marked with italic font. Applying the k-tails algorithm with $k = 2$ (a discussion on choosing k follows below) leads to the FSM in Figure 3b. Next, we make the FSM deterministic and minimize it, which gives Figure 3c. Note that the markings of transitions are propagated; all italic transitions in Figure 3c originate from the reference specification. Finally, we compute m and m_t: There are 19 marked transitions, that is, $m = 19$ (multiple labels at one arrow count as multiple transitions). When computing the intersection of the mined specification in Figure 1b and Figure 3c, we use three of the marked transitions, that is, $m_t = 3$. Thus, the recall is $3/19 = 16\%$.

As the example illustrates, using the k-tails algorithm helps to find otherwise non-obvious similarities between FSMs. In contrast, an exact comparison of FSMs often cannot identify

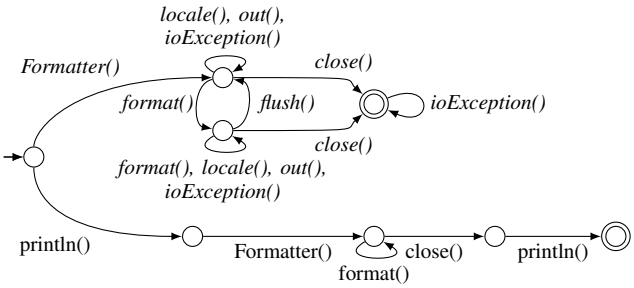


(a) Almost complete mined specification.

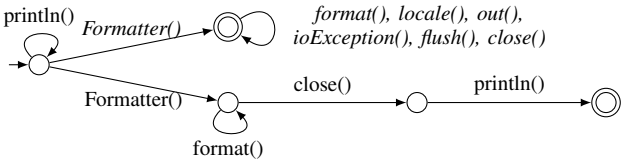


(b) Incomplete mined specification.

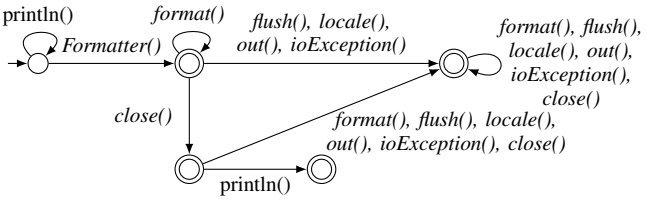
Figure 2: Mined specifications for *Formatter*.



(a) Union of mined specification and reference specification.



(b) Result of applying k-tails algorithm to (a).



(c) Deterministic and minimal union of (b).

Figure 3: Computation of recall using our metric with $k = 2$. Transitions printed in italics originate from the reference specification.

the similarity of two states because this similarity is hidden in non-matching parts of the FSMs. Instead, using k-tails allows for merging states that are equivalent within the limited visual range of the next k transitions. As a result, our metrics can find states to be similar even if they are not exactly equivalent, which leads to more reasonable recall and precision values than an exact comparison, in particular for incomplete and imprecise specifications.

The parameter k of the k-tails algorithm controls how much imprecision and incompleteness our metrics accept. A smaller k merges more states, leading to a less precise FSM

that accepts a more general language. Hence, our metrics detect more similarities but risk to consider states to be equivalent that incidentally share the same k-tail. In contrast, a larger k leads to a more precise FSM, that is, our metrics tolerate less imprecision and incompleteness. Since there is no perfect k, we compute recall and precision for $k = 1, 2, 3$ and without using k-tails (that is, comparing FSMs exactly), and average the results to get final values for recall and precision. We empirically found this solution to be practical for comparing FSMs. For illustration, the following table gives precision and recall computed with our metrics for the mined specifications in Figures 2a, 2b, and 1b.

Mined FSM	Precision					Recall				
	k=1	k=2	k=3	Exact	Avg.	k=1	k=2	k=3	Exact	Avg.
Figure 2a	100%	100%	100%	100%	100%	86%	86%	86%	77%	84%
Figure 2b	100%	100%	100%	100%	100%	43%	43%	43%	38%	42%
Figure 1b	30%	60%	0%	0%	23%	20%	16%	0%	0%	9%

III. Mining API Usage Specifications

This section presents three approaches to mine FSMs that describe ordering constraints of API methods. We compare the effectiveness of the mining approaches in Section IV using the evaluation framework presented in Section II. The miners dynamically analyze existing programs that use an API and summarize their API usage. The input to each miner is a method trace, that is, a sequence of method calls observed during program execution. At first, the miners extract *subtraces*, that is, subsequences that each contain calls relevant for a particular API usage and, ideally, do not include irrelevant calls. Then, these subtraces are summarized into FSMs.

Extracting subtraces from a large method trace is one of the most challenging aspects of dynamic specification mining, because typical method traces intermingle different API usages and contain many irrelevant method calls. The three mining approaches discussed in this section use different techniques to extract subtraces. First, we use a previously presented, method-centric approach. It is method-centric in the sense that it extracts subtraces by focusing on calls issued within a particular method. Second, we present a novel, object-centric approach, which we devised as an improvement of the first approach. The miner is object-centric in the sense that it focuses on calls centered around one object. It includes calls to a main object but also to other, related objects. Third, we use a simplified version of the object-centric approach, called single-object, which extracts subtraces that each contain calls to a single API object. In contrast to the object-centric approach, no calls to other objects are considered.

To gather method traces to be used as input to our miners, we instrument programs that use a particular API. An instrumented programs writes all method call and method return events in a log file. Concretely, we log the object identities and types of caller, callee, arguments, and the return value (if any), as well as the name of each called method. Our instrumentation only includes calls (returns) from (to) the instrumented program and ignores calls within libraries. In the following, we detail the three mining approaches.

A. Method-centric Specification Mining

The method-centric specification mining approach extracts subtraces by focusing on method calls issued within the execution of a certain method. As a first abstraction, all methods called within a given method are extracted. Then, the subtrace is further refined, for example, by filtering calls based on packages and by merging objects on which the same set of methods is called.

After extracting subtraces from a method trace, the miner summarizes them into FSMs. At first, similar subtraces are grouped together. The miner considers two subtraces to be similar of they involve objects on which the same sets of methods are called. For example, two subtraces involving a collection on which *iterator()* is called and an iterator on which *hasNext()* and *next()* are called are grouped together. Then, all similar subtraces are mapped into an FSM by creating a state for each called method and creating a transition between two states if two methods are observed consecutively. This technique for summarizing subtraces into FSMs is used by all three miners we evaluate in this paper. An example of the summarization follows in Section III-B.

B. Object-centric Specification Mining

To illustrate the object-centric mining approach consider Figure 4. Figure 4a shows a method that downloads data from a given URL using the *java.net* API. Executing the method produces a trace of method calls that intermingles several API usages, such as the trace on the left of Figure 4b.

To extract subtraces that each contain calls relating to a particular API class, we extract *object usage traces* from the method trace. Each object usage trace contains calls centered around a particular API object, but also other related calls. An object usage trace includes the following information:

- A core object o.
- All calls to o.
- For each parameter p of a call to o:
  - The (constructor) call that returned p.
  - All calls on p before it is passed to o.
- For each return value r of a call to o:
  - All calls on r after its return from o.

The miner creates an object usage trace for each instance of an API class on which at least two methods are called; instances with only one call cannot contribute any constraints between the methods of the API class.

Figure 4b illustrates the extraction of object usage traces. The miner creates three traces of calls centered around the instances of *URL*, *BufferedReader*, and *HashMap* respectively. Note that an object usage trace can contain calls to methods of different API classes. For example, the second object usage trace on the right of Figure 4b contains a call to *InputStreamReader()*, which shows the origin of the parameter to *BufferedReader()*.

During our experiments with the Java standard library, we found several API classes that should be specified together in one FSM. A typical example is the interplay of a collection

```
public class Downloader {
  private HashMap<String, Integer> history;
  public void download(String address) {
    URL url = new URL(address);
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(url.openStream()));
    String rawData = reader.readLine();
    while (rawData != null) {
      rawData = reader.readLine();
    }

    int number = 0;
    if (history.containsKey(address))
      number = history.get(address);
    history.put(address, number + 1);

    reader.close();
  }
}
```

(a) Different API usages that are intermingled. Calls to API methods are highlighted.

One execution trace:

URL()
URL.openStream()
InputStreamReader()
BufferedReader()
BufferedReader.readLine()
BufferedReader.readLine()
HashMap.containsKey()
HashMap.put()
BufferedReader.close()

Three object usage traces:

URL()
URL.openStream()

InputStreamReader()
BufferedReader()
BufferedReader.readLine()
BufferedReader.readLine()
BufferedReader.close()

HashMap.containsKey()
HashMap.put()

(b) Method trace from executing *download()* and object usage traces extracted by the object-centric miner.

Figure 4: Extracting object usage traces from method traces.

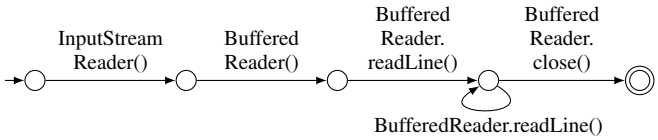


Figure 5: FSM mined from Figure 4 using the object-centric approach.

class, such as *LinkedList*, and its *Iterator*s. Specifying these classes in isolation cannot express constraints such as that a *LinkedList* must not be modified while iterating over it. Object usage traces can contain calls to several related objects, so that specifications involving multiple classes, such as *LinkedList* and *Iterator*, can be mined.

After extracting object usage traces for all instances of API classes from a method trace, the miner summarizes them into FSMs using the same technique as the method-centric miner. For example, the object usage trace for the instance of *BufferedReader* in Figure 4 results in the FSM in Figure 5.

C. Single-object Specification Mining

The single-object mining approach is a variation of the object-centric approach. For each instance of an API class, we extract a subtrace containing all calls to this instance but no other calls. Similarly to the object-centric approach, we only consider API objects on which at least two methods are called. For example, the subtraces extracted from the trace on the left of Figure 4b would be the same as given on the right of Figure 4b, except that the call to *InputStream()* is missing.

The part of the miner that summarizes subtraces into FSMs is shared with the other two approaches. As a result of considering calls to a single object in each subtrace, each mined specification only covers a single class. Hence, the single-object approach cannot mine specifications of method ordering constraints involving multiple classes, such as the interplay between a collection and its iterators.

IV. Experiments

The following section describes results from using our evaluation framework. We formalize specifications of 32 classes from the Java standard library and use them as a reference for evaluating specifications mined from twelve Java programs with the three mining approaches presented in Section III. Our experiments show that MCGs help to formalize FSMs that describe method ordering constraints of complex APIs. Furthermore, we show the evaluation framework to provide insights into the benefits of different mining approaches. We also use the framework to analyze the relation between the quality of method traces and the quality of mined specification from the traces. The results show that recall is bound by the coverage of an API in method traces. Finally, we compare our metrics to other metrics and a manual grading of mined specifications.

Table II lists the programs we analyze during our experiments and the number of runtime events generated by them. Overall, we analyze 47 million runtime events. We analyze programs from different domains, including several networking programs. The reason for selecting networking programs is that we focus our experiments on the *java.net* and *java.util* APIs. Most of the traces result from running unit tests or benchmark suites. Three programs are run as part of the DaCapo benchmark suite [30]. Programs for which no automated execution was possible were executed manually. The last column of Table II gives the origin of the traces.

A. Creating Reference Specifications with MCGs

We select 32 widely used classes from the *java.net* and *java.util* API and formalize their method ordering constraints using the approach described in Section II-A. For each class, we extract information on method ordering constraints from the corresponding API documentation (Javadoc) and a widely used reference book [31]. We describe these constraints using MCGs. Table III shows the number of methods of each class and the number of groups required to summarize the

Program	Description	Runtime events	
Eclipse	Integrated development environment	498,328	B
luindex	Text indexing tool	1,511,632	B
PMD	Source code analyzer	295,750	B
Diego	MP3 streaming server	10,482	M
JGroups	Reliable Multicast Communication	10,100,253	T
jo!	Web server	29,091,779	T
OpenCHAT	Chat server	84,591	M
Soap-Stone	Network benchmarking application	4,146,130	B
Testingpoint	Server monitoring	142,081	M
Voldemort	Distributed database	82,888	T
XBrowser	Web browser	435,575	M
Yafta	FTP client	142,061	M
Sum		46,541,550	

Table II: Programs and number of events analyzed for each. The last column states whether traces were obtained by running benchmarks (B), unit tests (T), or by executing the program manually (M).

API class/interface	MCGs		FSM	
	Methods	Groups	States	Transitions
java.net.DatagramSocket	28	15	7	80
java.net.MulticastSocket	15	10	5	34
java.net.Socket	41	23	14	207
java.net.URL	16	5	5	57
java.net.URLConnection	43	14	21	318
java.util.ArrayDeque	36	7	6	49
java.util.ArrayList	32	17	41	407
java.util.Collection	15	7	6	24
java.util.Deque	36	7	6	49
java.util.EnumMap	17	8	7	42
java.util.EnumSet	18	7	6	27
java.util.Formatter	7	8	4	13
java.util.HashMap	17	8	7	42
java.util.HashSet	18	7	6	27
java.util.Hashtable	20	8	7	48
java.util.IdentityHashMap	17	8	7	42
java.util.LinkedHashMap	17	8	7	42
java.util.LinkedHashSet	18	7	6	27
java.util.LinkedList	51	17	41	465
java.util.List	30	17	41	405
java.util.Map	17	8	7	42
java.util.NavigableMap	38	8	7	84
java.util.NavigableSet	29	7	6	42
java.util.PriorityQueue	20	7	6	29
java.util.Queue	19	7	6	28
java.util.Set	18	7	6	27
java.util.SortedMap	23	8	7	54
java.util.SortedSet	24	7	6	33
java.util.StringTokenizer	6	6	4	14
java.util.TreeMap	38	8	7	84
java.util.TreeSet	29	7	6	42
java.util.WeakHashMap	17	8	7	42
Average	24.1	9.2	10.2	91.4

Table III: Reference specifications with sizes of method constraint groups and the corresponding finite state machines.

constraints between these methods. On average, the constraints between 24 methods are represented in 9 groups.

Using our framework, we translate the MCGs into FSMs. The last two columns of Table III show the size of the generated FSMs in terms of their number of states and transitions. All FSMs are minimized, and hence, can be considered to be the most concise possible representation of the method ordering constraints. On average, the reference specifications contain 10 states and 91 transitions. In view of the 9 groups required to encode the same specifications, we conclude that MCGs are an effective help to formalize FSMs.

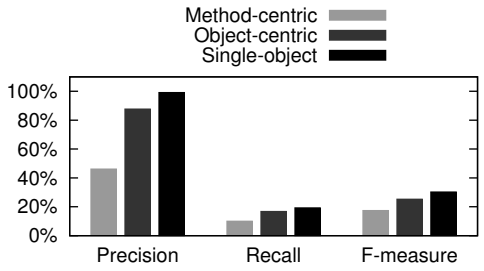


Figure 6: Comparison of three mining approaches.

B. Evaluating Specification Miners

We use the 32 reference specifications and the metrics presented in Section II-B to evaluate the three specification miners presented in Section III. For each miner, we compute precision and recall of all mined specifications with respect to all reference specifications. In addition to precision and recall, we compute the *F-measure*, the harmonic mean of precision and recall, which is typically used to combine precision and recall into a single value. Then, we select for each reference specification the mined specification with maximum F-measure, that is, the mined specification that is closest to the reference. For 15 out of 32 reference specifications, a specification with non-zero F-measure is mined; for the other 17, no matching specification can be mined, because the specified API class is not used in the method traces that we use as input.

Figure 6 shows the average precision, recall, and F-measure achieved using the three mining approaches. The results are averaged over all reference specifications with a non-null F-measure. The method-centric miner achieves an F-measure of 18%. The object-centric miner improves the F-measure to 25%, with a precision of 88%. The single-object miner achieves the best results with a precision of 99%, a recall of 19%, and an F-measure of 30%. The high precision results from considering calls to only one class per FSM, which avoids including any irrelevant calls by construction for the prize of not mining specifications involving multiple classes. The recall achieved by all three miners is rather low (below 20%). As we discuss in Section IV-C, low recall can be partly attributed to the quality of the traces we analyze. Yet, there seems potential to improve the recall in all three mining approaches.

As the results show, our evaluation framework helped us to significantly enhance the method-centric miner. Having appropriate metrics to assess the quality of mined specifications allowed us to achieve a relative improvement over the method-centric miner of 73% (for the F-measure).

Besides the miners described in Section III, we experiment with other mining approaches. For example, we adapt the FSM summarization of the object-centric and the single-object approach in such a way that all subtraces with a core object of the same class are summarized into one FSM. That is, more subtraces go into a single FSM, providing the opportunity to cover more API usages, while risking to include more irrelevant call sequences. Our evaluation framework confirms this reasoning: The specifications mined with this approach have higher recall but lower precision.

C. Influence of API Coverage

We analyze how the coverage of an API in a set of method traces influences the results of mining specifications from these traces. For each reference specification, we compute the *specification coverage* as the proportion of methods in the specification that are called at least once in the analyzed traces.

Figure 7 shows precision and recall depending on the specification coverage of each reference specification. The results show that recall is bound by coverage, which confirms the intuition that the quality of a dynamic specification miner cannot exceed the quality of the analyzed traces. The closer the recall achieved by a miner is to the diagonal displayed in the lower part of Figure 7, the more of the available information is extracted by the miner. Our results show that the single-object miner gets closer to this boundary then the other mining approaches, and hence, makes best use of the coverage provided by the traces. In contrast to recall, there is no clear relation between precision and coverage. This result is not surprising, since low precision results from considering method calls that are not relevant for a specification, and hence, do not affect coverage.

Interpreting our results, one should be aware that our notion of coverage does not fully reflect the quality of method traces. For example, a trace containing all required API method calls in random order has 100% coverage, but is not useful for mining correct specifications. Nevertheless, one can assume that most parts of a program use an API correctly, and hence, larger coverage coincides with more relevant call sequences that a miner can learn from.

D. Comparison with Other Metrics

It is an interesting question how well our metrics and previously proposed ones match an intuitive notion of the quality of mined specifications. To answer it, we manually assign precision and recall values to a set of mined FSMs and compare this manual grading to our metrics and those of [23] and [32]. We randomly select 18 mined FSMs and compare it to the reference specification of *StringTokenizer*, which is chosen because sufficiently many FSMs are mined for it. Among the 18 mined specifications, there are six from each miner. Five of them contain at least one call to *StringTokenizer* (that is, they are related to the reference specification), while one has no call to *StringTokenizer* at all (that is, it is obviously unrelated). To manually grade the mined FSMs we use the following procedure: For the precision value, we divide the

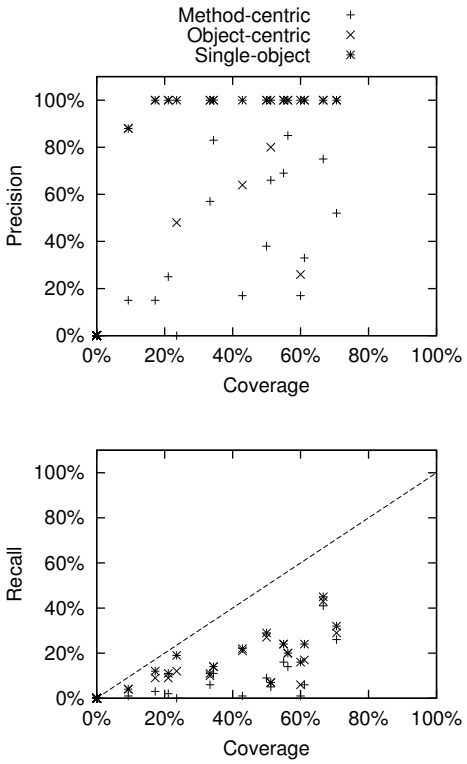


Figure 7: Precision and recall depending on coverage of the API by the analyzed traces.

number of transitions that are labeled with a method occurring in the reference specification and that are in a "correct" position by the total number of transitions. For the recall value, we describe the reference specification by a set of simple temporal properties, such as "n() must occur after m()" or "m() can be called multiple times". Then, we divide the number of temporal properties expressed in the mined specification by the total number of temporal properties.

Figure 8 shows precision, recall, and F-measure of the different metrics against the manually assigned values. Quante and Koschke's metric gives a single similarity value, which we compare to the F-measure. Linearity expresses that a metric matches the manual grading. For all three values, our metric is closest to the manually assigned values. Furthermore, the figures illustrate that Quark's metric often gives 0% or 100%, which is caused by not considering imprecision and incompleteness of mined specifications. Quante and Koschke's metric gives non-zero similarities even for mined specifications that are obviously unrelated to the reference.

The correlation between manual grades and the metrics confirms our interpretation of Figure 8:

	Our metric	Quark	Quante et al.
Precision	91%	54%	-
Recall	74%	58%	-
F-measure/Similarity	86%	44%	37%

Our metric correlates more to the manual grading than previously proposed metrics and provides acceptable correlations

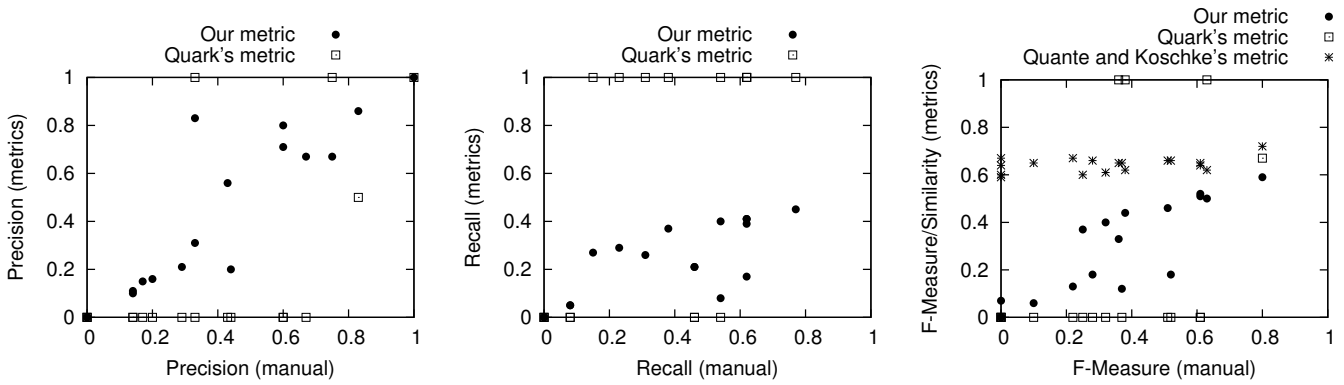


Figure 8: Comparison of three metrics with respect to manual grading of 15 mined specifications. Linearity indicates high agreement of a metric and the manually assigned values.

for all three values. We conclude that our metric matches an intuitive notion of the quality of mined specifications.

V. Related Work

A. Specification Mining

Existing specification miners can be grouped into miners of method ordering constraints and miners of other specifications. Many miners of the first group mine FSMs: Ammons et al. pioneered to mine specifications of method call sequences from dynamic execution traces using a probabilistic FSM learner [1]. SMArTIC is a specification mining framework that structures the mining process into trace filtering, trace clustering, probabilistic FSM learning, and merging of FSMs [8]. 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 [15]. Ghezzi et al. present a dynamic analysis to synthesize FSMs using graph transformations [17]. Other miners producing FSMs or regular expressions include [2], [4], [9], [11], [13], [20], and [21]. Our evaluation framework is applicable to all of the above mining approaches. Apart from FSM miners, there are approaches for mining method ordering constraints into other formats, such as pairs of related methods [5], [7], [14] and extended FSMs [16].

B. Evaluation of Specification Miners

Lo and Khoo propose a framework for empirically assessing automaton-based specification miners [23]. Their approach generates traces from a given FSM R and feeds them to a specification miner. The FSMs produced by the miner are then compared to R. Precision and recall are computed as the proportion of traces from one FSM that are accepted by the other (and vice versa). The approach is appropriate to evaluate how accurately a miner summarizes traces that include a well-defined set of method calls. However, it ignores extracting subsequences of semantically related method calls from a large execution trace, which is one of the most challenging parts of dynamic specification mining. Our framework can evaluate the entire mining process and uses robust metrics to measure precision and recall. Lo and Khoo's metrics fail to find similarities between two FSMs if the traces of one are not accepted by the other (and vice versa). Instead, our metrics account for imprecision and incompleteness in the mined specifications using the k-tails algorithm, and hence, can detect more similarities between FSMs.

Quante and Koschke propose a metric for the distance between FSMs, which is inspired by Levenshtein's string distance [32] (with a corrected version in [33]). Applying the distance metric to two FSMs gives a value d between zero and one. We compute the similarity as $1 - d$ to compare their metric with ours. Our way to compare a mined FSM to a reference improves over their approach in two ways. First, we provide two values, precision and recall, giving a more fine-grained estimate of how similar two FSMs are than a single value. Second, Quante et al.'s metric gives reasonable results for similar FSMs, but fails to identify dissimilarities. As shown in Section IV-D, even FSMs with disjoint alphabets can have a non-zero similarity. In contrast, our metrics are guaranteed to return precision and recall zero for disjoint FSMs.

Bogdanov and Walkinshaw propose an algorithm to structurally compare two FSMs [24]. The algorithm computes pairs of states (one from each FSM) with similar incoming and outgoing transitions and expresses the difference between the FSMs in terms of added and removed transitions. Their metrics are based on a purely structural comparison of FSMs, whereas our metrics compare the languages accepted by two FSMs.

Several approaches evaluate the quality of mined specifications through manual classification in "valid" and "invalid" specifications [11], [12], [14], [16], [19]. This approach is subjective and gives results that are hard to reproduce and compare. Another approach to validate mined specifications is to manually compare them to some reference, such as informal documentation or manually formalized specifications [1], [2], [5], [20]. In contrast, our evaluation framework supports the automatic comparison with a set of reference specifications. Another evaluation approach is to show the usefulness of mined specifications for some task, such as finding bugs [5], [7], [13], [18]. Similarly, some approaches are validated by checking programs with the mined specifications and considering rarely violated specifications to be valid [6], [10].

The drawback of such approaches is that they do not allow for comparing the quality of mined specifications. Finally, one can evaluate specifications by comparing the results from mining different programs, assuming that commonly found specifications describe typical behavior [21].

C. Other Related Work

Cornelissen et al. [34] propose a methodology and a set of metrics for evaluating trace reduction techniques. Their methodology focuses on trace reduction for the purpose of visualization and mainly uses quantitative metrics, such as trace size reduction, whereas we focus on the qualitative assessment of mined specifications.

A problem related to comparing FSMs is measuring the similarity between graphs. Several approaches to error-correcting graph matching have been proposed [35], which try to match one graph to another while changing the graphs if necessary. Since FSMs are particular graphs, error-correcting graph matching algorithms can be used to compare FSMs. However, graph similarity measures can find FSMs to be similar that actually describe very different languages, because they do not consider the particular semantics of certain nodes, such as initial and final states.

VI. Conclusions

Specifications of method ordering constraints can support software maintenance tasks, such as testing and program understanding. Although various approaches to automatically mine such specifications have been proposed, no evaluation technique has been established that allows comparing different miners on a common ground. This paper presents an evaluation framework to systematically assess the quality of mined specifications describing method ordering constraints of APIs. The framework helps building a set of reference specifications and provides metrics to compare mined specifications to the references. We use the framework to evaluate three mining approaches and show their respective benefits and drawbacks. Using the framework helped us to significantly improve the mining technique of our previously presented miner.

Acknowledgments

Thanks to David Lo, Zoltán Majó, Mathias Payer, and the anonymous reviewers for their comments on this paper. Also thanks to Jochen Quante for clarifying questions on his metric.

References

[1] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *POPL*, 2002, pp. 4–16.
[2] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *ISSTA*, 2002, pp. 218–228.
[3] T. Xie and D. Notkin, "Automatic extraction of object-oriented observer abstractions from unit-test executions," in *ICFEM*, 2004, pp. 290–305.
[4] 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 *ICSM*, 2005, pp. 155–164.
[5] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *TACAS*, 2005, pp. 461–476.
[6] V. B. Livshits and T. Zimmermann, "DynaMine: Finding common error patterns by mining software revision histories," in *ESEC/FSE*, 2005, pp. 296–305.
[7] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *ICSE*, 2006, pp. 282–291.
[8] D. Lo and S.-C. Khoo, "SMArTIC: Towards building an accurate, robust and scalable specification miner," in *FSE*, 2006, pp. 265–275.
[9] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *WODA*, 2006, pp. 17–24.
[10] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *ESEC/FSE*, 2007, pp. 25–34.
[11] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," in *ISSTA*, 2007, pp. 174–184.
[12] J. Henkel, C. Reichenbach, and A. Diwan, "Discovering documentation for Java container classes," *IEEE T. Software Eng.*, vol. 33, no. 8, pp. 526–543, 2007.
[13] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *ESEC/FSE*, 2007, pp. 35–44.
[14] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *ICSE*, 2007, pp. 240–250.
[15] M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in *FSE*, 2008, pp. 339–349.
[16] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *ICSE*, 2008, pp. 501–510.
[17] C. Ghezzi, A. Mocci, and M. Monga, "Synthesizing intensional behavior models by graph transformation," in *ICSE*, 2009, pp. 430–440.
[18] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *ESEC/FSE*, 2009, pp. 383–392.
[19] C. Le Goues and W. Weimer, "Specification mining with few false positives," in *TACAS*, 2009, pp. 292–306.
[20] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *ASE*, 2009, pp. 307–318.
[21] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *ASE*, 2009, pp. 371–382.
[22] C. J. Van Rijsbergen, *Information Retrieval*. Butterworths, 1979.
[23] D. Lo and S.-C. Khoo, "QUARK: Empirical assessment of automaton-based specification miners," in *WCRE*, 2006, pp. 51–60.
[24] K. Bogdanov and N. Walkinshaw, "Computing the structural difference between state-based models," in *WCRE*, 2009, pp. 177–186.
[25] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behaviour," *IEEE T. Comput.*, vol. 21, pp. 592–597, 1972.
[26] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *ESEC/FSE*, 2009, pp. 345–354.
[27] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *ACM T. Softw. Eng. Meth.*, vol. 7, no. 3, pp. 215–249, 1998.
[28] S. P. Reiss and M. Renieris, "Encoding program executions," in *ICSE*, 2001, pp. 221–230.
[29] J. E. Hopcroft and J. D. Ullman, *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.
[30] S. M. Blackburn *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*, 2006, pp. 169–190.
[31] D. Flanagan, *Java in a nutshell*. O'Reilly, 2005.
[32] J. Quante and R. Koschke, "Dynamic protocol recovery," in *WCRE*, 2007, pp. 219–228.
[33] J. Quante, "Dynamic object process graphs," Ph.D. dissertation, Universität Bremen, 2009.
[34] B. Cornelissen, L. Moonen, and A. Zaidman, "An assessment methodology for trace reduction techniques," in *ICSM*, 2008, pp. 107–116.
[35] H. Bunke, "Error correcting graph matching: On the influence of the underlying cost function," *IEEE T. Pattern Anal.*, vol. 21, no. 9, pp. 917–922, 1999.