Early Filtering of Polluting Method Calls for Mining Temporal Specifications

Hao Zhong, Lu Zhang
School of Electronics Engineering and Computer Science
Peking University, China
{zhonghao04, zhanglu}@sei.pku.edu.cn

Tao Xie
Department of Computer Science
North Carolina State University, USA
xie@csc.ncsu.edu

Abstract

Temporal specifications are critical in software maintenance, but they are often not provided due to the high cost of writing them manually or being out-of-date due to the rapid evolution of software. The lack of accurate specifications poses a barrier for maintainers to make changes because maintainers may misunderstand important properties of the programs under modification and alter these properties in unintended ways. As API clients contain many usages of libraries including temporal rules, various approaches have been proposed to automatically mine temporal specifications from these clients. Typically, only a small part of the mined specifications are real specifications because the generated traces from clients are quite large and polluted. In this paper, we analyze four types of method calls that can pollute the generated traces. Filtering these method calls after traces are generated takes extra efforts of not only trace collection but also trace mining. It is desirable that these polluting method calls are filtered out as early as possible. To address the problem, we develop a tool, called MATS (Mining Accurate Temporal Specifications), that can filter out most of the preceding polluting method calls before traces are generated. Our experiment shows that with these filtering techniques, the specifications mined by MATS are more accurate than without these filtering techniques. We also show the detailed impacts of MATS's filtering techniques and thresholds. The results provide further insight on how and why MATS can improve existing specification mining.

1 Introduction

A temporal specification defines the legal call sequences of a library. Temporal specifications are critical in many software maintenance activities including verifying programs against their intended behavior [12], understanding programs of their usages [28], and debugging programs to detect defects [26]. Without accurate specifications, maintainers may misunderstand important properties of the programs under modification and alter these properties in unintended ways. However, temporal specifications are seldom provided due to the high cost of writing them manually [4]. Even when temporal specifications are provided, as software evolves, these specifications are often out-of-date [9]. With software evolution, it is common that a new version of a library contains changes that can make the out-of-date specifications no longer valid to client code [10]. As a result, low coherence between programs and specifications poses a barrier for maintainers to make changes of programs.

An example temporal specification `open`$\rightarrow$`close` defines the legal call sequence that when client code is to access files using libraries, every call of `open` should be followed by a call of `close`. This specification is intuitive, but due to the complexity of libraries [22], many temporal specifications of libraries are not so straightforward. As existing clients exhibit many usages of libraries, various approaches [3, 20] have been proposed to mine temporal specifications from these clients. Before the mining process, traces are extracted from clients statically [11, 23] or dynamically [8, 29]. After that, temporal specifications can be mined by specification miners from the extracted traces. The existing mining approaches can be mainly divided into automata-based approaches [3, 7] and sequence-based approaches [20, 29]. Automata-based approaches can mine temporal specifications in the form of finite state automata (FSA), and sequence-based approaches can mine temporal specifications in the form of frequent call sequences (FCS).

After temporal specifications are mined, real specifications can be determined manually via inspection as Weimer and Necula [26] do. Cable developed by Ammons et al. [4] can also be used to debug a mined specification against existing traces to reduce the inspection effort. As shown by Weimer et al. [25, 26], after inspection, only 1% to 11% of the mined specifications are real specifications. They point out that the primary reason for inaccuracy of current specification mining is the lack of traces because it is difficult to extract sufficiently good-quality traces for mining specifications. Typically, the extracted traces are quite large and polluted. Shoham et al. [23] also complain that pure method calls pollute their traces and pose a barrier for their tool to be deployed as an industrial tool.

We develop a new approach and its supporting tool called MATS to prevent polluting method calls from being generated traces as early as possible for mining accurate specifications. This paper makes the major contributions as follows:

- We analyze four types of polluting method calls and develop MATS to filter out most of them before traces are generated.
- We conduct an experiment on six open source libraries. The result shows that with filters, MATS can produce more accurate specifications in less time than without filters.
- We conduct more experiments to evaluate the impacts of MATS's filters and thresholds. The results provide insight on why and how MATS can improve existing specification mining.

The remainder of this paper is organized as follows. Section 2 shows the four types of polluting method calls. Section 3 presents our approach. Section 4 shows our experiments. Section 5 discusses issues of our approach. Section 6 discusses related work, and Section 7 concludes.

2 Polluting Method Calls

This section presents when polluting method calls occur and how they can pollute the mined specifications.

Pure method calls. The pollution of pure method calls occurs when a library has public pure methods. From the perspective of a library, the pure methods are its declared methods that do not mutate its states [5]. From the perspective of client code, most pure methods of a library can be called anywhere and are in fact noises in the generated traces. To demonstrate how pure method calls can pollute mined specifications, we use a class of a library `sun.jdbc.odbc.JdbcOdbcConnection` as an example. Let us consider the three methods of the class: `close`, `open`, and `getURL`. Here, `getURL` is a pure method, and it simply returns the URL of a connection. If we consider the three methods equally of interest, the method calls of `getURL` are recorded in the extracted traces. Sometimes, the calls of `getURL` in a particular set of traces can follow some temporal rules, and the temporal rules may be summarized as temporal specifications. When we use the specifications with `getURL` to detect bugs in clients, false positives can be produced because the specification violations do not correspond to defects. From the preceding example, we observe that pure methods such as `getURL` can pollute the mined specifications. Shoham et al. [23] also complain that these noises limit their tool and plan to eliminate them in their future work.

Some pure methods of a library can return the states of the library, and the returned states can be used as guides to call other methods of the library. For example, let us consider two other methods of `sun.jdbc.odbc.JdbcOdbcConnection`: `commit` and `isClosed`. It is a good practice to call `isClosed` to check whether a collection is closed before every call of `commit` to prevent possible exceptions, although it may be too strict to maintainers because in practice, we can find many violations of this usage in existing clients. Filtering out all pure method calls can lose such strict specifications as `isClosed`→`commit`, but it can improve the accuracy of specification mining in general. This kind of pollution can be reduced by the library code analyzer of MATS (Section 3.1).

Internal method calls. The pollution of internal method calls occurs when traces are dynamically extracted from clients. To generate traces for a library of interest, we often instrument only the library instead of all the clients as it can save instrumentation effort. After a library is instrumented, it is easy to change the settings of clients to make these clients use the instrumented library. When we instrument a library, as we cannot determine which methods will be called, we usually instrument all the methods of the library. Thus, when client code invokes a method of the library, the method call and all its internal method calls are all recorded in the generated trace. The internal method calls can pollute the mined specifications because some internal methods (e.g., those private ones) cannot be directly invoked by client code and even when some internal methods (e.g., those public ones) can be invoked by client code, these methods invoked by client code may follow different usage patterns than the ones internally invoked by the library. To illustrate the situation, let us use the following code snippet from a class of a library `sun.jdbc.odbc.JdbcOdbcConnection` as an example.

```
public  Statement createStatement(int i, int j){
  ...
  jdbcodbcstatement.initialize(...);
  jdbcodbcstatement.setBlockCursorSize(...);
  registerStatement(...);
  return jdbcodbcstatement;
}
```

When client code calls `createStatement`, the corresponding extracted trace will be as follows.

```
  createStatement;
  ...
  initialize;
  setBlockCursorSize;
  registerStatement;
```

Here, for the ease of the discussion, we ignore the internal method calls of `initialize`, `setBlockCursorSize`, and `registerStatement`. Yang et al. [29] point out that the call relationships between `createStatement` and the three other methods can produce false specifications, which are less interesting to maintainers. After the specifications are mined, Yang et al. [29] use a reachability heuristic to filter out the mined frequent call pairs between `createStatement` and the other three methods. Although they consider that the remaining method calls are interesting to maintainers, actually these internal method calls can still cause to produce traces that do not exist in clients. In this example, the preceding call sequence cannot exist in clients because `registerStatement` is a protected method that is actually not accessible to client code. Besides a method's access modifier, other factors such as the modifications of a library's internal fields can also produce traces that do not exist in client code because internal methods (both public and private ones) can modify these fields whereas client code cannot. When we use the specifications mined from these traces to check clients, false positives can thus be produced. This kind of pollution can be reduced by the trace generation tool of MATS (Section 3.2).

Method calls within loops. The pollution of method calls within loops occurs when many methods of a library are called within *loop*-statements of client code. As some *loop*-statements can have many iterations, the method calls within these *loop*-statements can be recorded many times in the generated trace. When there are many method calls of this kind in the traces, the usages within *loop*-statements are quite possible to be mined as temporal specifications. If these mined specifications are used to detect bugs, they can also introduce false positives because these specifications have bias from *loop*-statements. This kind of pollution can be also reduced by the trace generation tool of MATS (Section 3.2).

Unrelated method calls. The pollution of unrelated method calls occurs when the methods of a specification are actually unrelated. For example, Weimer and Mishra [25] show a false specification: `java.util.Set.add`→`java.util.Timer.cancel`. This specification is a false one because its two method calls are unrelated. To reduce the pollution from unrelated method calls, existing approaches use various techniques to refine the mined specifications. Among these techniques, many are in fact heuristics and can work well only on particular libraries. For example, Yang et al. [29] use a name similarity heuristic to filter out the mined specifications whose method names are not similar. This heuristic can work well when method names follow a particular naming convention but may not work well when the method names do not follow it. For example, the following code snippet is from a class of a library `com.microsoft.jdbc.base.BaseConnection`.

```
public abstract class BaseConnection...{
  private BaseConnectionPool secondaryImplConnections;
  public final void open(...){
     ...
     secondaryImplConnections = new BaseConnectionPool();
  }

  public final synchronized void close(){
     ...
     secondaryImplConnections.close();
  }
}
```

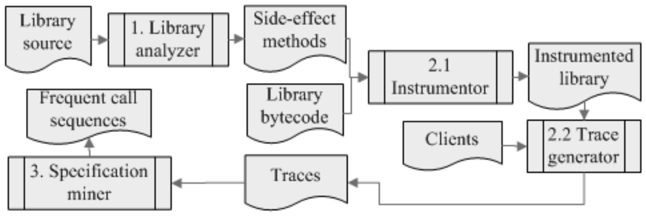

Figure 1. Overview of MATS

In this example, the specification `open`→`close` is a real one because violations may cause memory leak. According to the preceding name similarity heuristic, this specification can be filtered out because the similarity value between `open` and `close` is zero. Instead, MATS finds relevant methods of a library through their modified states. As a result, MATS can consider the preceding two methods relevant correctly because they both modify the state of `secondaryImplConnections`. Different from other filtering techniques of MATS, the corresponding filtering technique for this pollution is introduced after the mining process (Section 3.3). This filtering technique uses the results of the library code analyzer of MATS (Section 3.1).

3 Approach

Figure 1 shows the overview of MATS. It consists of a library-code analyzer (Section 3.1), a trace tool (Section 3.2), and a specification miner (Section 3.3).

3.1 Static Library Analysis

The library-code analyzer aims to find side-effect methods to be instrumented. MATS uses the side effect analysis proposed by Salcianu et al. [21] to find out the side-effect methods of a library. Besides finding out the side-effect methods, for every side-effect method, MATS records all its modified fields. These fields are further used to filter out unrelated methods of the mined frequent sequences. These unrelated methods do not modify the same field.

3.2 Trace Generation

The trace generation tool instruments the library and generates traces using our filtering techniques. JRat[1] is an open source profiling tool, and it is extensible to generate traces. For example, both Perracotta [29] and SMArTIC [18] extend JRat to dynamically extract traces from clients. MATS also extended JRat and added the following new features:

[1]`http://jrat.sourceforge.net/`

Instrumenting a subset of methods with callsite contexts. Instead of instrumenting all the methods of a library, MATS instruments only the side-effect methods identified by the library code analyzer. When a method is instrumented, the instructions to gather the called method name and its callsite contexts (e.g., methods in the call stack) are added to the bytecode. When the method is executed, the method name and the callsite contexts are sent to MATS. As the pure methods of libraries are not instrumented, they will not be recorded in the generate traces. For the internal method calls of libraries, MATS can filter them out by checking their callsite contexts.

Using method event queue to filter out the method calls within *loop*-statements. When we run client code with an instrumented library, the instrumented instructions can send the called method names of the library and their corresponding callsite contexts to MATS. To eliminate the pollution from method calls within *loop*-statements, MATS uses an event queue for all the incoming method call pairs. If a method call pair does not exist in the queue, MATS enqueues this method call pair and writes it in the generated trace file. If a method call pair exists in the queue, MATS simply ignores this method call pair. For the method calls within *loop*-statements, only the method calls of the iterations that can produce different method call sequences can be written into the generated trace files. Section 4.2 further discusses the impact of the queue's length on generating traces. As MATS's callsite contexts do not include the enclosing statements of method calls, it can also filter out repeated method calls outside *loop*-statements. As these repeated method calls may not exhibit extra usages for a library of interest, MATS filters out these repeated method calls for simplicity. In future work, we plan to record the start of the first loop iteration and the end of the last loop iteration of the same loop in the trace so that our techniques can precisely identify method calls invoked inside the same loop body

After filtering out polluting method calls, MATS groups the remaining method calls by threads and generates a separate file for each thread.

3.3 Specification Miner

The specification miner mines frequent call sequences from traces. In particular, MATS uses a sequence miner [5] to mine frequent call sequences. Sequence mining [2] has been the the core of various existing sequence-based mining approaches such as CHRONICLER [20]. We do not choose to mine automata from the traces because the problem of mining automata from traces comes down to the classical grammar inference problem [7], which is proved to be NP-complete [13]. Ammons et al. [3] also point out that it is impossible to find an automata that can exactly accept the traces to be learned. The imprecision may also pollute the mined specifications.

Library	Version	Time	Method	Side-effect
jfreechart	1.0.1	1188.4	6402	3681
jung	1.7.6	899.4	3472	2254
itext	2.0.7	2524.1	5470	3138
jdom	1.0.1	206.4	684	374
bcel	5.2	365.5	2118	1296
velocity	1.5	246.7	1405	844
Total		5430.5	19551	11587

Table 1. Results of library code analyzer

To mine frequent call sequences, MATS needs to find out relevant traces first. For every side-effect method of a library, MATS searches for the traces that include the method and encodes these traces into a transaction database (see [5] from details). A sequence miner can take the transaction database as an input and produce the frequent sequences whose supports are greater than a threshold. For a set of sequences (C), the support of a frequent sequence (s) is defined by existing sequece miners as follows:

$$support(s) = \frac{\#\ of\ sequences\ with\ s}{\#\ of sequences\ in\ C} \qquad (1)$$

From the definition, if $m_1 \to m_2 \to m_3$ is mined as a frequent sequence, $m_1 \to m_2$, $m_1 \to m_3$, and $m_2 \to m_3$ are also mined as frequent sequences. After removing the latter three redundant frequent sequences, MATS further uses the result of the library-code analyzer to filter out frequent sequences whose methods are all unrelated. For a mined frequent sequence $m_1 \to m_2 \to \ldots \to m_n$ and their corresponding modified fields F_1, F_2,. . ., F_n, MATS filters out a sequence if in the sequence there is no such methods m_i, m_j that can satisfy $F_i \cap F_j \neq \emptyset$ $(1 \leq i \leq j \leq n)$. After this filtering process, MATS turns the remaining frequent sequences into frequent call sequences.

4 Experiments

All the experiments were conducted on a PC with an Intel Pentium 2.26 GHz CPU and 1512 M memory running the Windows 2000 Professional operating system. Section 4.1 presents an experiment that uses MATS to mine specifications of five libraries with its filters on and off. Section 4.2 presents the impact of MATS's filtering techniques to generate traces. Section 4.3 presents the impact of the threshold of MATS. Section 4.4 presents the results when traces are statically extracted, and Section 4.5 discusses the threats to validity.

4.1 Mining Specifications

Analyzing library code. As introduced in Section 3.1, the library-code analyzer of MATS can find the side-effect

Library	All On		All Off		Client	
	Time	***Method Call***	***Time***	***Method Call***	***Number***	***LOC***
jfreechart	36.4	2242	5114.1	1748247	10	689085
jung	645.1	3032	35214.5	15073284	10	63919
itext	658.7	4052	20128.1	8832711	11	297182
jdom	1246.2	17497	90124.7	25111237	17	193223
bcel	1286.7	22004	32457.1	9458471	9	258460
velocity	21.1	553	39248.4	8426778	17	310350
Total	3894.2	49650	222286.9	68650728	74	1812219

Table 2. Generated traces

Library	All On					All Off				
	Time	***RS***	***TS***	***RS/TS***	***AL***	***Time***	***RS***	***TS***	***RS/TS***	***AL***
jfreechart	254.0	9	125	7.2%	6.41	110649.4	4	1125	0.3%	5.59
jung	348.4	19	61	31.1%	7.66	836745.1	5	824	0.6%	5.64
itext	408.7	83	376	22.1%	5.03	449264.7	11	928	1.2%	5.31
jdom	1057.1	12	206	5.8%	6.82	514897.4	6	1206	0.5%	6.5
bcel	1302.0	12	109	11.0%	3.90	559710.2	4	2758	0.1%	4.91
velocity	12.4	2	6	33.3%	5.83	188954.9	4	428	0.9%	4.09
Total	3382.7	137	883	15.5%	5.69	2660221.7	34	7369	0.5%	5.34

Table 3. Mined specifications

methods of libraries, and Table 1 shows the results. Column 1 lists the names of the libraries. Column 2 lists the version numbers of these libraries. Column 3 lists the time used to analyze the libraries in seconds. Column 4 lists the numbers of the total methods. Column 5 lists the numbers of the side-effect methods. We observe that about 50% to 60% of the total methods are side-effect methods. MATS instruments only the side-effect methods of a library, and thus prevents the pollution from its pure methods.

Generating traces. After each library was instrumented, their clients were set up and executed for collecting traces. Table 2 shows the results. Column "Library" lists the name of the libraries. Column "All On" lists the results when we turn on all the filters. Column "'All Off" lists the results when we turn off all the filters. Column "Client" lists the number and LOC of clients that are used to generate the traces. For the subcolumns, Columns "$Time$" list the times used to generate the traces in seconds. Columns "$Method\ Call$" list the numbers of the recorded method calls.

From Table 2, we make two observations. One is that although we use million lines of clients for traces, the numbers of recorded method calls of "All On" are relatively small. The other is that the numbers of recorded method calls of "All Off"are much larger than those of "All On". The numbers of recorded method calls of "All On" are relatively small giving the size of clients because it is quite common that only a small part of client code calls the library of interest. Although we use millions lines of clients, many method calls are not relevant to the library of interest, and these method calls are not recorded. The numbers of recorded method calls of "All Off"are much larger than those of "All On" because the filtering techniques of MATS successfully filter out many polluting method calls. From Table 2, we can see that about 99.9% method calls of "All Off" can be filtered out by these filters. As the remaining method calls are relatively small in size, MATS saves the effort to generate traces. We show the detailed impact of MATS's filtering techniques on the generated traces in the next section.

Weimer and Mishra [25] discuss the technical problems to generate accurate traces and point out that the lack of traces is the primary reason for the inaccuracy of specification mining. Our experiment reveals another practical problem: although we use million lines of clients for collecting traces, when the polluting method calls are filtered out, the numbers of remaining method calls are actually small.

Mining frequent call sequences. After the traces were generated, for every side-effect method, MATS searched for the traces that included the method and encoded them into the form of a transaction database. From the transaction database, MATS mined frequent call sequences and further refined the mined sequences using the relevant analysis technique in Section 3.3 for frequent call sequences. In this experiment, the threshold for a frequent call sequence was set to 0.8. We show the impact of threshold in Section 4.3.

Table 3 shows the details of the mined specifications. Column "All On" lists the results when we turn on all the filters. Column "All Off" lists the results when we turn off all the filters. For the subcolumns, Columns "$Time$" list the times used to mine the specifications in seconds. As the remaining method calls are small in size, MATS also saves the effort to mine specifications. Columns "RS" list the numbers of the real specifications. As were determined by Weimer and Necula [26], these real specifications are manually determined via inspection. The violations of a real

Library	Description of Usage	All On		Side-effect		Callsite		Queue		All Off
		M	*E*	*M*	*E*	*M*	*E*	*M*	*E*	*All*
jfreechart	draw a bar chart	10	99.893%	3734	60.213%	89	99.052%	6784	27.714%	9385
	draw a pie chart	6	99.814%	749	76.840%	72	97.774%	300	90.724%	3234
jung	create a network	45	99.988%	19926	94.515%	8591	97.635%	203	99.944%	363259
	create a graph	30	98.555%	455	78.083%	64	96.917%	364	82.466%	2076
itext	create a pdf table	11	99.097%	872	28.407%	28	97.701%	112	90.805%	1218
	create an image pdf file	7	99.982%	37299	4.285%	43	99.890%	78	99.800%	38969
jdom	save an XML file	8	99.951%	4067	74.982%	280	98.278%	110	99.323%	16256
	validate an XML file	3	99.748%	519	56.350%	16	98.654%	88	92.599%	1189
bcel	parse a jar file	104	99.842%	6846	89.614%	36772	44.232%	643	99.025%	65937
	obfusticate a jar file	76	99.643%	6424	69.846%	7769	63.533%	454	97.869%	21304
velocity	create a java file	3	99.998%	115400	30.487%	6	99.996%	813	99.510%	166013
	create another java file	5	99.968%	12478	20.253%	42	99.732%	396	97.469%	15647
Total		308	99.956%	208771	70.366%	53772	92.367%	10345	98.532%	704487

Table 4. Impacts of MATS's filtering techniques

Library	Description of Usage	All On		Side-effect		Callsite		Queue		All Off
		T	*E*	*T*	*E*	*T*	*E*	*T*	*E*	*All*
jfreechart	draw a bar chart	0.1	0.5%	12.1	56.6%	0.2	0.8%	184.1	865.4%	21.3
	draw a pie chart	0.2	1.0%	5.8	35.1%	0.2	1.0%	102.5	619.8%	16.5
jung	create a network	16.1	1.8%	37.6	4.3%	23.6	2.7%	10944.1	1262.2%	867.1
	create a graph	0.9	11.8%	5.5	68.6%	1.7	21.8%	32.2	402.9%	7.9
itext	create a pdf table	0.2	2.9%	4.8	88.6%	0.3	6.2%	14.4	263.2%	5.5
	create an image pdf file	0.2	0.1%	135.8	96.7%	0.3	0.2%	1782.6	12693.8%	140.4
jdom	save an XML file	0.6	1.3%	11.9	25.4%	1.9	4.0%	11934.9	25480.0%	46.8
	validate an XML file	0.3	8.2%	1.7	43.5%	0.3	7.9%	16.3	415.9%	3.9
bcel	parse a jar file	7.8	5.1%	31.1	20.3%	90.5	59.0%	2817.8	1835.0%	153.6
	obfusticate a jar file	2.7	2.7%	59.9	61.4%	21.4	22.0%	1204.9	1235.8%	97.5
velocity	create a java file	0.2	0.1%	8611.8	84.9%	0.5	0.1%	244777.5	5110.1%	484.9
	create another java file	0.2	0.3%	130.7	63.3%	0.2	0.3%	6826.1	14100.5%	48.4
Total		29.2	1.5%	748.5	39.5%	141.1	7.5%	76680.5	4048.9%	1893.9

Table 5. Impacts of MATS's filtering techniques (Cont.)

specification indicate defects in client code. Columns "TS" list the numbers of the total mined specifications. Columns "AL" list the average lengths of corresponding "TS". From the values of "*RS/TS*", we observe that MATS's filtering techniques improve the mining results effectively. Below are 5 frequent call sequences mined by MATS with the filters on. For each frequent call sequence, the bracket shows the modified fields of its involved methods.

```
1:com.lowagie.text.Document.open
->com.lowagie.text.pdf.PdfContentByte.beginText
->com.lowagie.text.pdf.PdfContentByte.endText
->com.lowagie.text.Document.close (open, close,content)

2:com.lowagie.text.pdf.PdfStamper.createSignature
->com.lowagie.text.pdf.PdfStamper.close (sigApp, stamper)

3:edu.uci.ics.jung.visualization.VisualizationViewer.
suspend->edu.uci.ics.jung.visualization.Visualization-
Viewer.unsuspend (model)

4:edu.uci.ics.jung.visualization.FRLayout.initialize
->edu.uci.ics.jung.visualization.FRLayout.lockVertex
->edu.uci.ics.jung.visualization.FRLayout.unlockVertex
(dontmove)

5:org.jdom.input.SAXHandler.startEntity
->org.jdom.input.SAXHandler.endEntity (entityDepth,
inInternalSubset, suppress)
```

4.2 Impact of Filtering Techniques

This section shows the impact of MATS's filtering techniques on trace generation. To generate traces, for each library, we prepared two different usages to generate traces with MATS's different filters on. We chose different usages of all these libraries to reduce possible bias in the experiment, and Table 4 shows the result. Column "Library" lists the names of the libraries. Column "Description of Usages" lists the descriptions of the prepared usages. Columns "All On", "Side-effect", "Callsite", "Queue", and "All Off" list the results when we use the corresponding filtering technique to generate traces of given usages. We later use these titles to denote their corresponding filtering techniques. In particular, "Side-effect" filters out only pure method calls but not other polluting method calls. "Callsite" filters out only internal method calls. "Queue" filters out only method calls within *loop*-statements. "All On" uses all these filtering techniques in a sequential order as MATS does. "All Off" turns off all the preceding filtering techniques. For the subcolumns, Columns "M" list the numbers of collected method calls using a corresponding filter-

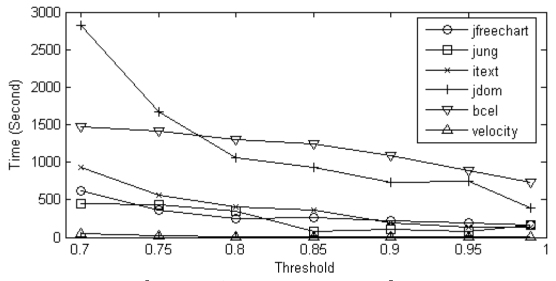

Figure 2. Impact on time

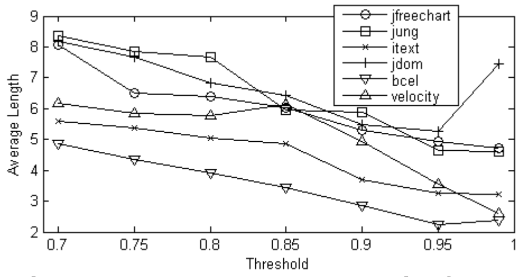

Figure 3. Impact on Avg. Length of FCS

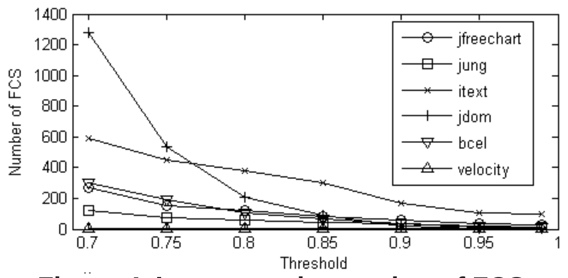

Figure 4. Impact on the number of FCS

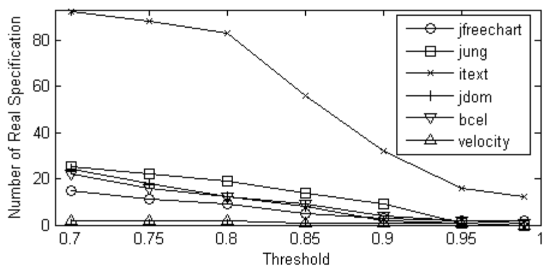

Figure 5. Impact on the number of real specification

ing technique. Columns "All" list the numbers of collected method calls of "All Off". Columns "E" are calculated as $(1 - M/All) \times 100\%$ correspondingly.

From the results of Table 4, "Side-effect" can reduce about 60% to 90% method calls for the usages of libraries such as *jfreechart*, *jung*, *jdom*, and *bcel*. For *itext* and *velocity*, "Side-effect" can reduce only about 20% method calls. "Callsite" can reduce about 90% method calls for the libraries except *bcel*. For *bcel*, "Callsite" can reduce about 50% method calls. "Queue" works well with all the usages except one usage of *jfreechart*. In this usage, "Queue" can reduce about 27.7% method calls. The different performances of these filtering techniques are dependent on how the libraries are implemented and how they are used by clients. As shown in "All On", when all the preceding three filtering techniques are turned on, about 99.9% method calls can be reduced.

Table 5 shows the time used to generate traces. For the subcolumns, Columns "T" list the time used to generate a trace using the corresponding filtering technique in seconds. Columns "All" list the times used to generate a trace of "All Off" in seconds. Columns "E" are calculated as $T/All \times 100\%$ correspondingly. In general, for "Side-effect" and "Callsite", it takes less time than "All Off" does. For "Side-effect", pure methods are not instrumented, so the time to write pure method calls to the trace files is saved. For "Callsite", it takes less time to compare the callsite context of a method call than to write the method call to the trace files. As shown in Table 4, many method calls are internal method calls, which are compared but not written to the traces. As a result, it also saves the time to generate the trace files. For "Queue", it takes more time to generate the trace files than "All Off" does because the method calls are enqueued and checked with the previous method calls of the queue. In the current implementation of MATS, the length of the queue is set to 2000. We tried the queues of different lengths such as 30000, 20000, 10000, 5000, and 2000. The time used to generate traces are quite different, but the numbers of method calls of the generated traces are all the same. The numbers do not change because "Queue" is used to filter out the method calls within *loop*-statements, and the usages listed in Table 4 do not call more than 2000 methods of a library within a single *loop*. Not to call too many methods of a library within a single *loop* may reflect how programmers develop client code to use libraries. We do not further try queues whose lengths are smaller than 2000 because as shown in "All On", it is already fast enough. For "All On", "Side-effect", "Callsite", and "Queue" are used in a sequential order. As "Side-effect" and "Callsite" can filter out most of the method calls, the remaining method calls for "Queue" are in fact few. As a result, it also takes less time for "All On" to generate traces than "All Off" does. Totally, it takes about only 1% time of "All Off" to generate traces.

In summary, as MATS can effectively filter out the polluting method calls discussed in Section 2. The three filtering techniques can filter out 99.9% method calls and thus can also reduce the size of generated traces. These filters can not only prevent the mined specifications from pollution but also save time to generate traces.

4.3 Impact of Threshold

As shown in Section 3.3, MATS uses a sequence miner to mine specifications. The threshold used by a sequence miner may have impact to its results. In this section, we used different thresholds for frequent call sequences and

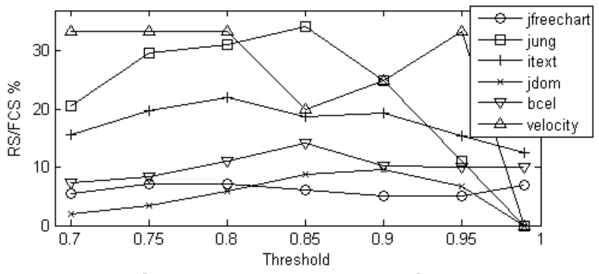

Figure 6. Impact on RS/FCS

discuss the impact of the threshold. Because for sequence miners, a low threshold can hurt its performance significantly [15], we used relatively high thresholds to rerun MATS's specification miner in this experiment.

Figures 2 to 6 show the results. In these figures, the horizontal axes all denote thresholds. The vertical axis of Figure 2 shows the impact of thresholds on time. We observe that with the increase of the threshold, it takes less time for MATS to mine frequent call sequences for all the libraries. The vertical axis of Figure 3 shows the average lengths of the mined frequent call sequences. In general, with the increase of the threshold, the average lengths of the frequent call sequences become smaller. The vertical axis of Figure 4 shows the number of the mined frequent call sequences. The vertical axis of Figure 5 shows the number of the real specifications. With the increase of the threshold, both values decrease, but the the number of frequent call sequences decreases faster. To show the best ratio of the preceding two values, we further build Figure 6 from Figures 4 and 5. The vertical axis of Figure 6 shows the value of the number of the real specifications to the number of the total mined frequent call sequences. From Figure 6, we can see that for most libraries, the max values occur when the threshold is between 0.8 to 0.85.

In summary, the threshold of the sequence miner has the impact as follows. With the increase of the threshold, the time, the average length of frequent call sequence, the number of frequent call sequences, and the number of real specifications all decrease. For most libraries, we can get the best ratio between the number of real specifications to the number of frequent call sequences when the threshold is between 0.8 to 0.85.

4.4 Comparison with Static Approaches

Section 4.1 shows that the filters of MATS can improve the accuracy of the mined specifications with less effort when traces are dynamically extracted. In this section, we compare the accuracy of the mined specifications with approaches that statically extract traces. To conduct the comparison, we implemented a source code analyzer based on Eclipse's JDT compiler [1] to extract traces statically from the clients listed in Table 2. After the traces were extracted, they were fed to the specification miner of MATS. The threshold for a frequent call sequence was set to 0.8.

Library	Filters			No filters		
	RS	*TS*	*RS/TS*	*RS*	*TS*	*RS/TS*
jfreechart	6	257	2.3%	4	446	0.9%
jung	9	128	7.0%	5	135	3.7%
itext	29	226	12.8%	19	407	4.7%
jdom	10	189	5.3%	8	1415	0.6%
bcel	8	173	4.6%	7	2837	0.2%
velocity	3	25	12.0%	2	22	9.1%
Total	65	998	6.5%	45	5262	0.9%

Table 6. Comparison of static approaches

Table 6 shows the results. Column "Filters" lists the results with the filters of MATS. Column "No Filters" lists the results without the filters of MATS. For the subcolumns, Columns "RS" list the numbers of the real specifications. Columns "TS" list the numbers of the total mined specifications. The results of "Filters" are better than the results of "No Filters", but it is not as significant as shown in Table 3. When traces are statically extracted, it is relatively easy to analyze client code only. Consequently, the extracted traces are not polluted by internal method calls. In addition, for every *loop*-statement, we extract its enclosing method calls only once. Consequently, the pollution from method calls within *loop*-statements is also reduced. The remaining filters can still improve the results but not so significantly. For examples, the following false specification is mined with the filters off. With the filters on, the involved methods can be filtered out because they are all pure methods.

```
1:org.apache.bcel.generic.InstructionHandle.getNext
->org.apache.bcel.generic.InstructionHandle.getInstruction
->org.apache.bcel.classfile.Code.getMaxLocals
```

From Tables 3 and 6, we observe that all the approaches perform relatively well on libraries such as *jung* and *itext*, and all perform relatively not well on libraries such as *jfreechart*, *jdom*, and *bcel*. This observation may reflect the reality that some libraries are more restricted by temporal rules than other libraries. Without the filters of MATS, the results of static approaches are better than the results of dynamic approaches because the extracted traces are less polluted by internal method calls and method calls within *loop*-statements. With the filters of MATS, the results of dynamic approaches are better than the results of static approaches because polluting method calls in dynamically extracted traces are effectively filtered out by MATS's filters, whereas the statically extracted traces are polluted by some other factors such as infeasible paths that are not handled by MATS yet.

4.5 Threats to Validity

The internal threats of our experiment include that the real specifications are manually determined via inspections, which needs human efforts as what was done by Weimer and Necula [26]. Another way to measure the accuracy of the mined specifications is to feed all the mined specifications to a verification tool to check their violations for bugs. As many mined specifications are false ones, our preliminary experiment shows that the number of violations is typically quite huge. It takes even more human efforts to determine whether these violations are bugs or not than to inspect real specifications. Yang et al. [29] and Ramanathan et al. [20] present a possible solution by using a few selected real specifications for detecting bugs. To automate the process, we still need to improve the accuracy of the mined specification by eliminating more polluting factors.

The external threats of our experiment include that other specification miners also have techniques to filter out some mined specifications. Although these filters are introduced after the specifications are mined and they take extra efforts, they can still improve the accuracy of the mined specifications. To make a full comparison of all these techniques, we need to take all these into consideration and to conduct more experiments.

5 Discussion

In this paper, we focus on the effectiveness of MATS's filtering techniques to improve sequence-based approaches of mining specifications from both dynamically and statically extracted traces. Automata-based approaches can also be affected by polluting method calls. For example, Shoham et al. [23] complain that their automata-based approach is limited by pure method calls. Ammons et al. [3] also point out that some traces contain method call sequences that cannot be accepted by an automata learner. Lo and Khoo [18] cluster traces first. Before a trace cluster is fed to the automata learner, they iteratively remove traces that cannot be accepted by the automata learner. As the filters of MATS can filter out many polluting method calls, it may also improve the results of automata-based approaches. We plan to conduct more experiments to assess MATS's effectiveness on automata-based approaches in future work.

6 Related Work

Extracting traces. Existing clients exhibit many valuable usages of libraries including temporal rules. Before the mining process, traces are extracted from these clients statically [20, 23] or dynamically [3, 29]. As pointed out by Weimer and Mishra [25], the low quality of extracted traces is the primary reason for the inaccuracy of current specification mining. MATS aims to address this problem and uses different filtering techniques to filter out polluting method calls for mining accurate specifications. As the extracted traces are often large in size, research on compressing the traces has also been done. For example, STEP developed by Brown [6] can compress a raw trace to reach 5% of its size. MATS can also reduce the size of the generated traces significantly, but MATS is specially developed for specification mining. MATS is different in purpose with STEP.

Mining specification. The existing mining approaches can be mainly divided into automata-based approaches and sequence-based approaches.

For automata-based approaches, Strauss developed by Ammons et al. [3] uses an extended Angluin algorithm [19] to mine automata from the traces that are related by traditional dataflow dependencies. SMArTIC developed by Lo and Khoo [18] further improves Strauss by introducing clustering techniques to refine traces before the mining process. Kremenek et al. [14] use an automata template to find methods that fit the automata template. Whaley et al. [27] mine automata-like models from traces. Dallmeier et al. [8] present an approach to extract predicates and to mine named automata from traces. Cook and Wolf [7] reduce the general problem of mining automata from traces to the classical grammar inference problem. The problem has been proved to be NP-complete [13]. Ammons et al. [3] also point out that it is impossible to find an automata that can exactly accept the traces to be learned. The preceding limitations may compromise the accuracy of the mined specifications. MATS has no such limitations because the underlying learner of MATS is a frequent sequence miner.

For sequence-based approaches, Eagler et al. [11] use the z-statistic value as a support value to mine frequent call sequences. Yang et al. [29] use method-call-pair templates to mine the method calls whose supports are greater than a threshold into frequent call sequences. Li and Zhou [16] use frequent itemset mining to extract implicit programming properties and detect their violations for detecting bugs. DynaMine developed by Livshits and Zimmermann [17] mines properties from software revision histories. Wasylkowski et al. [24] use sequence mining for frequent call sequences to detect anomalies in clients. Ramanathan et al. [20] use sequence mining for frequent call sequences from the statically extracted traces. These approaches use raw traces as an input, and these traces are often quite large and noisy. MATS develops various techniques to filter out polluting method calls, complementing these approaches. When polluting method calls are filtered out, the mined specifications can be more accurate.

Filtering. One practical way to refine mined specifications is to filter out false ones. Weimer and Necula [26] present a set of heuristics to keep a mined method pair $a \rightarrow b$ that satisfies all the criterions as follows: (1) The pair $a \rightarrow b$ exists in an exceptional control flow; (2) There must be at least one error trace with *a* but without *b*; (3) a and b are in the same package; (4) Every value and receiver object expression in *b* must also be in *a*. Yang et al. [29] use a reachability heuristic and a name similarity heuristic to refine the mined sequences. Most of the filtering techniques take extra efforts because they are applied when specifications are already mined. MATS filters most of the polluting methods before traces are generated; thus, MATS can save the analysis time effectively. In addition, most of the preceding filtering techniques are heuristics and may work well in particular libraries whereas MATS's filtering techniques are not.

7 Conclusion

Temporal specifications are critical in many software maintenance activities. However, temporal specifications are often not provided because of the high cost of writing them manually or being out-of-date because of software evolution, posing a barrier for maintainers to make changes. To automatically mine these specifications from existing clients, many approaches have been proposed, but in practice, only a small part of the mined specifications are real specifications because traces are typically polluted. In this paper, we analyze four types of polluting method calls and develop MATS to filter out polluting method calls before traces are generated. To assess the effectiveness of MATS's filters, we conduct an experiment on MATS with its filters on and off to mine specifications for six open source libraries. The results show that with the filters, MATS can produce more accurate specifications in less time. We also conduct experiments on MATS's different filtering techniques and thresholds, and its applications on static approaches. The results provide insight on why MATS can improve existing specification mining.

References

[1] M. Aeschlimann, D. Baumer, and J. Lanneluc. Java tool smithing extending the Eclipse Java Development Tools. In *EclipseCon*, 2005.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. ICDE*, pages 3–14, 1995.

[3] G. Ammons, R. Bodík, and J. Larus. Mining specifications. In *Proc. POPL*, pages 4–16, 2002.

[4] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging temporal specifications with concept analysis. In *Proc. PLDI*, pages 182–195, 2003.

[5] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proc. KDD*, pages 429–435, 2002.

[6] R. H. F. Brown. STEP: A framework for the efficient encoding of general trace data. Master's thesis, McGill University, Montréal, Québec, Canada, 2003.

[7] J. Cook and A. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[8] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Proc. WODA*, pages 17–24, 2006.

[9] S. Deelstra, M. Sinnema, and J. Bosch. Experiences in software product families: Problems and issues during product derivation. In *Proc. SPLC*, pages 165–182, 2004.

[10] D. Dig and R. Johnson. The role of refactorings in API evolution. In *Proc. ICSM*, pages 389–398, 2005.

[11] D. Engler, D. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. SOSP*, pages 57–72, 2001.

[12] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proc. ASE*, pages 412–416, 2001.

[13] E. Gold. Complexity of automatic identification of given data. *Information and Control*, 10:447–474, 1978.

[14] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proc. OSDI*, pages 259–272, 2006.

[15] S. Laxman and P. Sastry. A survey of temporal data mining. *Sadhana*, 31(2):173–198, 2006.

[16] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. ESEC/FSE*, pages 306–315, 2005.

[17] V. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proc. ESEC/FSE*, pages 31–40, 2005.

[18] D. Lo and S. Khoo. SMArTIC: towards building an accurate, robust and scalable specification miner. In *Proc. ESEC/FSE*, pages 265–275, 2006.

[19] A. Raman and J. Patrick. The sk-strings method for inferring PFSA. In *Proc. Machine Learning Workshop Automata Induction, Grammatical Inference, and Language Acquisition*, 1997.

[20] M. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proc. ICSE*, pages 240–250, 2007.

[21] A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. *Proc. VMCAI*, pages 199–215, 2005.

[22] C. Scaffidi. Why are APIs difficult to learn and use? *Crossroads*, 12(4):4–4, 2005.

[23] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proc. ISSTA*, pages 218–228, 2007.

[24] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. ESEC/FSE*, pages 35–44, 2007.

[25] W. Weimer and N. Mishra. Privately finding specifications. *IEEE Transactions on Software Engineering*, 34(1):21–32, 2008.

[26] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proc. TACAS*, pages 461–476, 2005.

[27] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. ISSTA*, pages 218–228, 2002.

[28] T. Xie and J. Pei. MAPO: mining API usages from open source repositories. In *Proc. MSR*, pages 54–57, 2006.

[29] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proc. ICSE*, pages 282–291, 2006.