Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications

Mithun Acharya[1], Tao Xie[1], Jian Pei[2], and Jun Xu[1,3]
[1]Department of Computer Science, North Carolina State University, USA
[2]School of Computing Science, Simon Fraser University, Canada
[3]Google Inc., USA
{acharya, xie}@csc.ncsu.edu, jpei@cs.sfu.ca, junxu@csc.ncsu.edu

ABSTRACT

A software system interacts with third-party libraries through various APIs. Using these library APIs often needs to follow certain usage patterns. Furthermore, ordering rules (specifications) exist between APIs, and these rules govern the secure and robust operation of the system using these APIs. But these patterns and rules may not be well documented by the API developers. Previous approaches mine frequent association rules, itemsets, or subsequences that capture API call patterns shared by API client code. However, these frequent API patterns cannot completely capture some useful orderings shared by APIs, especially when multiple APIs are involved across different procedures. In this paper, we present a framework to automatically extract usage scenarios among user-specified APIs as partial orders, directly from the source code (API client code). We adapt a model checker to generate interprocedural control-flow-sensitive static traces related to the APIs of interest. Different API usage scenarios are extracted from the static traces by our scenario extraction algorithm and fed to a miner. The miner summarizes different usage scenarios as compact partial orders. Specifications are extracted from the frequent partial orders using our specification extraction algorithm. Our experience of applying the framework on 72 X11 clients with 200K LOC in total has shown that the extracted API partial orders are useful in assisting effective API reuse and checking.

Categories and Subject Descriptors: F.3.1 [Specifying and Verifying and Reasoning about Programs]: Specification techniqes

General Terms: Verification.

Keywords: Mining, Specificaiton, Usage Scenarios, Partial Orders, API Patterns

1. INTRODUCTION

A software system interacts with third-party libraries through various APIs. Using these library APIs often needs to follow certain usage patterns. Furthermore, ordering rules (specifications) exist between APIs, and these rules govern the secure and robust operation of the system using these APIs. But these patterns and rules may not be well documented by the API developers. Then

it is difficult for system developers to effectively or correctly reuse these APIs during system development or verify the correct usage of these APIs after the system has been built. To address the problem, previous approaches have mined likely API usage patterns from software systems that reuse APIs. Some approaches [15–18, 23, 25] exploit the static program information extracted from system source code, whereas other approaches [4, 5, 27] exploit the dynamic program information extracted from system executions, which require setup of runtime environments and availability of sufficient system tests.

API usage patterns recovered by most of these previous approaches are in the form of frequent association rules, itemsets, or subsequences. Association rules [2,17,18] characterize pairs of API calls that are often used together without considering their orders. Frequent itemsets [10, 15] characterize sets of API calls that are often used together without considering their orders. Frequent subsequences [22, 26] characterize sequences of API calls that are often used together while considering their orders. Although these mined API usage patterns have been shown to be useful to some extent, they cannot completely capture some useful orderings shared by APIs, especially when multiple APIs are involved across different procedures.

To address the issues faced by previous approaches in mining API usage patterns, we develop a framework to automatically extract frequent partial orders among user-specified APIs, directly from the source code (API client code). Frequent partial orders summarize important ordering information from sequential patterns (common API usage scenarios). Frequent partial orders provide more information about the ordering than the sequential patterns while providing more general and more concise API ordering information. The mined partial orders among APIs can assist the correct and effective API reuse by the programmers. The partial orders can also assist in inferring API ordering rules (specifications) that govern the secure and robust operation of the system using these APIs.

This paper makes the following main contributions.

Static API Trace Generation: We adapt a model checker to generate interprocedural control-flow-sensitive static traces related to the APIs of interest. Our techniques allow mining of open source systems that reuse the APIs of interest without requiring environment setup for system executions or availability of sufficient system tests.

Scenario Extraction: A single static trace from the model checker might involve several API usage scenarios, being often interspersed. We present an algorithm to separate different usage scenarios from a given trace, so that each scenario can be fed separately to the miner.

API Partial Order Mining: We present novel applications of a miner in mining partial orders among APIs from static API traces.

```
#include <abcdef.h>
void p ( ) { b ( ); c ( ); }
void q ( ) { c ( ); b ( ); }
void r ( ) { e ( ); f ( ); }
void s ( ) { f ( ); e ( ); }

int main ( )
  {
  int i, j, k;
  a ( );
  if ( i == 1 )
    {
    f ( ); e ( ); c ( );
    exit ( );
    }
  else
    {
    if ( j == 1 )
      p ( );
    else
      q ( );
    d ( );
    if ( k == 1 )
      r ( );
    else
      s ( );
    }
  }
```

(a) Example code

1	a → f → e → c
2	a → b → c → d → e → f
3	a → c → b → d → e → f
4	a → b → c → d → f → e
5	a → c → b → d → f → e

(b) Static program traces

```
a → b → d → e
a → b → d → f
a → c → d → e
a → c → d → f
```

(c) Frequent sequential patterns
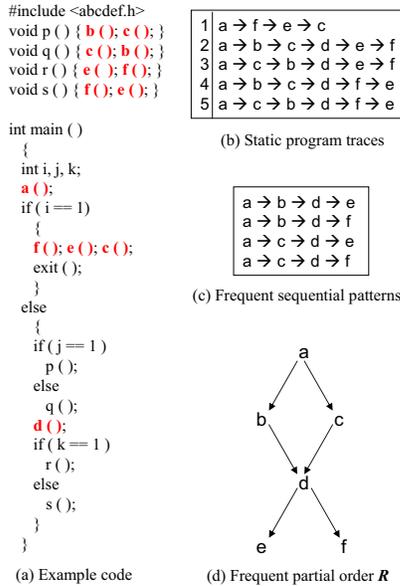
(d) Frequent partial order **R**

Figure 1: A simple example illustrating our framework

The mined partial orders provide important, useful API ordering information that is not provided by patterns mined by previous approaches.

Specification Extraction: We present an algorithm to effectively extract real specifications from mined partial orders. Our approach separates real specifications (that are universally true for a given set of APIs) from frequent partial orders that are only specific to the clients used for mining, thereby reducing the number of reported false specifications.

Implementation and Experience: We describe an implementation of the framework by adapting a publicly available model checker called MOPS [7] and adopting a miner called FRECPO [19]. We apply our framework on 72 X11 clients with 200K LOC in total and compare our approach with an existing specification miner. Our results highlight the unique benefits of our approach and show that the extracted API partial orders are useful in assisting effective API reuse and checking.

The remainder of this paper is structured as follows. Section 2 starts with an example that motivates our framework. Section 3 introduces the formal framework for mining API partial orders, and describes the various components in our framework in detail. Section 4 presents the implementation details. Section 5 reports our evaluation experience. Section 6 discusses related work. Section 7 discusses limitations of our framework and future work. Finally, Section 8 concludes.

2. EXAMPLE

This section illustrates a partial order that our framework extracts for a given set of related APIs, directly from the source code. Figure 1(a) shows a simple code snippet in C that uses APIs from a header file <abcdef.h>, namely, a, b, c, d, e, and f. Suppose that a programmer wants to investigate whether there are some usage orders among the APIs from <abcdef.h>.

Figure 1(b) shows five program traces involving these APIs along different possible control paths in the program. Given a support threshold min_sup, a sequential pattern is a sequence s that appears as subsequences of at least min_sup sequences. For example, let min_sup be 4. The four sequences shown in Figure 1(c) are sequential patterns since they are subsequences of Sequences 2,

3, 4, and 5 (all except a → f → e → c). Sequential patterns capture the frequent call patterns shared by program traces. However, the four sequential patterns cannot completely capture the ordering shared by APIs a, b, c, d, e, and f. It is easy to see that the partial order R shown in Figure 1(d) is shared by the four program traces. We can make the following interesting observations from the partial order R:

- The partial order R summarizes the four sequential patterns: the four sequential patterns are paths in the partial order R. Note that the only sequences with a support greater than 4 are a → f, a → e, and a → c, each with a support of 5.

- The partial order R provides more information about the ordering than the sequential patterns. For example, R indicates that b and c are called in any order, but often before d. Hence the mined partial order R effectively summarizes the sequential patterns among APIs and provides more general and more concise API ordering information to the programmers.

- If the min_sup is sufficiently high, the partial order provides strong hints on likely specifications that should be true for the correct operation of the program. For example, if the partial order R was mined from traces with a very high support, then with high confidence, "d should always follow a along any path" is a specification that should be satisfied by all programs using the APIs a and d.

This example motivates the idea of using frequent partial orders to effectively summarize sequential patterns among APIs and provide more general and more concise ordering information to the programmers. However, there are many issues not obvious in the motivating example, and these issues shall be addressed throughout the paper. (1) In general, if T is the set of all traces along all execution paths in the program, then T is an uncomputable set. Furthermore, the length of a trace can be infinite. (2) Some of the generated traces might be infeasible in the program. (3) Along some program paths, APIs might not be used correctly. (4) A given trace might have more than one scenario involving APIs from <abcdef.h>, being all jumbled up (for example, a → a → b → c could be two separate scenarios, a → b and a → c, instead of one). Each scenario has to be extracted separately before being fed to the miner. (5) Partial orders with high support might just be a frequent usage pattern specific to the client code, and not a specification. We have to separate real specifications from false ones. We address these issues in the next section, where we present our framework for mining partial orders from static program traces.

3. FRAMEWORK

In this section, we formalize the notions introduced in the previous section. We define partial order, total order, and frequent closed partial order (FCPO) [19], and formalize the problem of mining frequent closed partial orders from program traces. After the necessary foundations have been laid, we present the various components of our framework: trace generator, scenario extractor, and specification extractor. We conclude this section by providing a complexity analysis of our framework components.

3.1 Partial and Total Order

A *partial order* is a binary relation that is reflexive, antisymmetric, and transitive. A *total order* (or called linear order) is a partial order R such that for any two items x and y, if $x \neq y$ then either $R(x, y)$ or $R(y, x)$ holds.
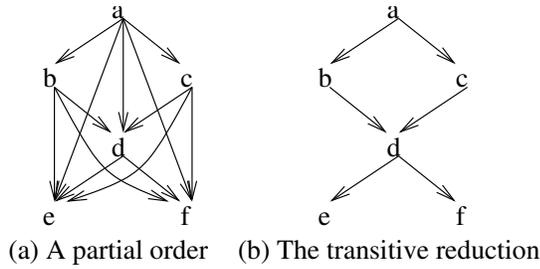
(a) A partial order   (b) The transitive reduction

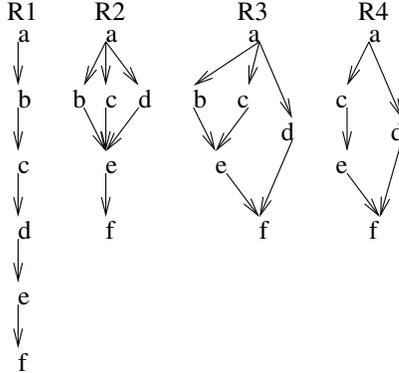**Figure 2: A partial order and its transitive reduction**

Figure 3: Four orders $R_1 \supset R_2 \supset R_3 \supset R_4$

A partial order R can be expressed in a Directed Acyclic Graph (DAG): the items are the vertices in the graph and $x \rightarrow y$ is an edge if and only if $(x,y) \in R$ and $x \neq y$. We also write an edge $x \rightarrow y$ as (x,y) or xy. For example, Figure 2(a) shows a partial order R, which has 13 edges.

Since a partial order is transitive, some edges can be derived from the others and thus are redundant. For example, in Figure 2(a), edge $a \rightarrow d$ is redundant given edges $a \rightarrow b$ and $b \rightarrow d$. Generally, an edge $x \rightarrow y$ is *redundant* if there is a path from x to y that does not contain the edge. For a partial order R, the *transitive reduction* of R can be drawn in a *Hasse diagram*: for $(x,y) \in R$ and $x \neq y$, x is positioned higher than y; edge $x \rightarrow y$ is drawn if and only if the edge is not redundant. Figure 2(b) shows the transitive reduction of the same partial order R in Figure 2(a). The transitive reduction has only 6 edges. For an order R, the transitive reduction may have much fewer edges.

In this paper, we draw a partial order in a Hasse diagram, i.e., its transitive reduction, and omit the isolated vertices. For example, Figure 3 shows four partial orders R_1, R_2, R_3, and R_4; R_1 is further a total order.

Let V be a set of items, which serves as the domain of our string database. A *string* defines a global order on a subset of V. A string can be written as $s = x_1 \cdots x_l$, where $x_1, \ldots, x_l \in V$. l is called the *length* of string s, i.e., $len(s) = l$. For strings $s = x_1 \cdots x_l$ and $s' = y_1 \cdots y_m$, s is called a *super-string* of s' and s' a *sub-string* of s if (1) $m \leq l$ and (2) there exist integers $1 \leq i_1 < \cdots < i_m \leq l$ such that $x_{i_j} = y_j$ ($1 \leq j \leq m$). We also say s contains s'. For a string database SDB (set of strings), the *support* of a string s, denoted by $sup(s)$, is the number of strings in SDB that are super-strings of s.

The *transitive closure* of a binary relation R is the minimal transitive relation R' that contains R. Thus, $(x,y) \in R$ provided that there exist $z_1, \ldots, z_n$ such that $(x, z_1) \in R$, $(z_n, y) \in R$, and $(z_i, z_{i+1}) \in R$ for all $1 \leq i < n$.

The total order defined by string $s = x_1 \cdots x_l$ can be written in the *transitive closure* of s, denoted by $\mathcal{C}(s) = \{(x_i, x_j) | 1 \leq i < j \leq l\}$. Note that, in the transitive closure, we omit the trivial pairs (x_i, x_i). For example, for string $s = abcd$, $len(s) = 4$. The transitive closure is $\mathcal{C}(s) = \{(a,b), (a,c), (a,d), (b,c), (b,d), (c,d)\}$. Here, we omit the trivial pairs (a,a), (b,b), (c,c), and (d,d).

The *order containment relation* is defined as, for two partial orders R_1 and R_2, if $R_1 \subset R_2$, then R_1 is said to be *weaker* than R_2, and R_2 is *stronger* than R_1. Intuitively, a partially ordered set (or poset for short) satisfying R_2 will also satisfy R_1. For example, in Figure 3, $R_4 \subset R_3 \subset R_2 \subset R_1$. Note that R_4 covers fewer items than the other three partial orders. Trivially, we can add the missing items into the DAG as isolated vertices so that every DAG covers the same set of items. To keep the DAG simple and easy to read, we omit such isolated items.

3.2 Frequent Closed Partial Orders (FCPO)

A *string database* SDB is a multiset of strings. For a partial order R, a string s is said to *support* R if $R \subseteq \mathcal{C}(s)$. The *support of R in SDB*, denoted by $sup(R)$, is the number of strings in SDB that support R. Given a minimum support threshold min_sup, a partial order R is called *frequent* if $sup(R) \geq min\_sup$. Following the related definitions and the order containment relation, we have the following result: for a string database SDB and partial orders R and R' such that $R' \subset R$, we have $sup(R') \geq sup(R)$. Therefore, if R is frequent, then R' is also frequent. To avoid the triviality, instead of reporting all frequent partial orders, we can mine the representative ones only.

Let us consider the program traces in Figure 1 again. The four sequential patterns can be regarded as frequent partial orders, which are supported by Traces 2, 3, 4, and 5. As discussed before, given that the partial order R is also supported by Strings 2, 3, 4, and 5, the four sequential patterns as frequent partial orders are redundant. There does not exist another partial order R' such that R' is stronger than R in Figure 1 and is also supported by Strings 2, 3, 4, and 5. In other words, R is the strongest one among all frequent partial orders supported by Strings 2, 3, 4, and 5. Thus, the partial order R is not redundant and can be used as the representative of the frequent partial orders supported by Strings 2, 3, 4, and 5. Technically, R is a frequent closed partial order.

A partial order R is *closed* in a string database SDB if there exists no partial order $R' \supset R$ such that $sup(R) = sup(R')$. A partial order R is a *frequent closed partial order* if it is both frequent and closed. We next formalize the process of mining FCPOs from program traces.

3.3 Formalizing FCPO Mining from Program Traces

Informally, our framework mines FCPOs for the APIs specified by the user from the program source code. Our framework addresses the following problems:

- Generating sequences of API invocations along different program paths. These sequences are stored as a string multiset database. However, generating all traces along all execution paths is an uncomputable problem and a trace can be of infinite size. Furthermore, a generated trace can be infeasible.

- Finding the complete set of frequent closed partial orders from the API sequence database with respect to a minimum support threshold min_sup.

Formally, let Σ be the set of valid program statements in the given program source code. Let $\mathcal{A}$ be the set of APIs specified by
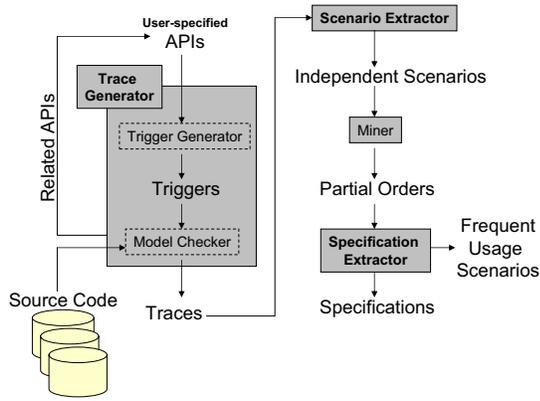
Figure 4: Framework for mining usage scenarios and specifications

Figure 5: A property FSM. end1 and end2 are final states

Figure 6: Trigger FSM that accepts the regular language $e(a_1 + a_2 + ... + a_k)^* x$

the user. A *trace* $t \in \Sigma^*$, a sequence of statements executed by a path p, is *feasible* if path p is feasible in the program. Let $T \subset \Sigma^*$ be the set of all feasible traces in the program. To simplify the definitions, let us assume that all APIs in $\mathcal{A}$ are empty methods, do not take any arguments, and return `void`[1]. For a given $t \in T$, let $A(t) \in \mathcal{A}^*$ be the API invocations along the trace t expressed as a string. $A(t)$ can be an empty string if t does not have any invocation of APIs from the set $\mathcal{A}$. Let $T' \subseteq T$ be the set of all feasible traces such that if $t \in T'$, $A(t)$ is not empty. However, the set T' is uncomputable and $t \in T'$ can be of infinite size. Our framework initially does the following steps:

- Generate the computable approximation of T' from the program and extract $A(t)$ for all t in the approximate set. The extracted $A(t)$'s are stored in a string database, say, V.

- Extract the set of FCPOs among APIs in $\mathcal{A}$ from V, with respect to a minimum threshold min_sup.

The high-level overview of our framework is shown in Figure 4. Our framework has three main components: trace generator, scenario extractor, and specification extractor. The user specifies a set of APIs from which *Triggers* (explained in Section 3.4) are generated. The Triggers are then used with push-down model-checking for trace generation. This process also recommends more APIs related to the user-specified set of APIs. The scenario extractor extracts various API scenarios (explained in Section 3.5) from the traces, which are then fed to a FCPO miner [19]. The mined partial orders are used by the specification extractor to output specifications. Each process is described in detail in the subsections below. We first explain the process of trace generation.

3.4 Trace Generation

We first briefly summarize the Push-Down Model Checking (PDMC) process [6,9], which we adapt for trace generation. To generate API invocation sequences along different program paths, we introduce

---

[1]By assuming that APIs are empty methods that do not take any arguments and return `void`, we restrict the program statements related to APIs from $\mathcal{A}$ in the program to only direct API invocations. If this is not the case, for a given API, say `a`, we have to consider all the program statements that affect and are affected by `a` by some data flow dependencies. For example, if we have `i = a(j); if (i != NULL ) b(i);`, then the program statements `if (i != NULL)` and `b(i)` are related to `a`. We relax this assumption to a certain extent later in the paper by incorporating simple data flow analysis.
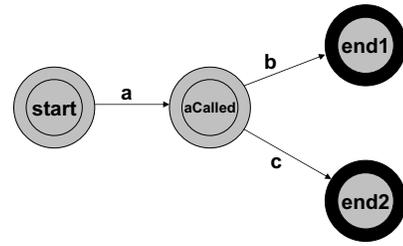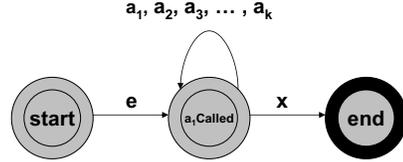
the concept of $Triggers$. Finally, we discuss the soundness of our approach.

3.4.1 Push-Down Model Checking (PDMC)

Informally, given a property represented using a Finite State Machine (FSM), PDMC [9] checks to see if there is any path in the program that puts the FSM in its final state. For example, if the property FSM is specified as shown in Figure 5, PDMC reports all program paths in which `a` is followed by either `b` or `c`. PDMC models the program as a Push Down Automata (PDA) and the property as an FSM. PDMC then combines the program PDA and the property FSM to generate a new PDA; the new PDA is then model checked to see if any *final configuration* in the PDA is reachable. A *configuration* of a PDA $\mathbf{P}$ is a pair $c = \langle q, \omega \rangle$, where q is the state in which the PDA is in and ω is a string of stack symbols in the PDA stack at that state. A configuration is said to be a final configuration, if q belongs to the set of final states in the FSM. If a final configuration is reachable, PDMC outputs the paths (in the program) that cause the resultant PDA to reach this final configuration. The resulting trace can either be feasible or infeasible because of data-flow insensitivity (being incomplete). However, if there is a program trace that puts the FSM in the final state, PDMC reports it (being sound). We next describe our *Triggers* technique that adapts PDMC to generate API invocation sequences in a program.

3.4.2 Triggers

Our goal is to generate the set $T' \subseteq T$ from the program and extract $A(t)$ for all $t \in T'$, $A(t) \in \mathcal{A}^*$, $\mathcal{A} = \{a_1, a_2, a_3, ..., a_k\}$, where $a_i, 1 \leq i \leq k$, are the APIs specified by the user. Let us assume that we give the FSM shown in Figure 6 to PDMC to be verified against a program $\mathcal{P}$. The FSM in Figure 6 accepts any string of the form $e(\sum_{i=1,2,...,k} a_i)^* x$, where e and x are any two points in the program. Given this Trigger FSM, PDMC outputs all program paths that begin with e and end with x in the program. By setting e as the entry point of the `main` routine and x as any exit point in the program, we can collect sequences of API along paths that begin at the `main` routine and ends at exit points.

Let $B \subseteq \Sigma^*$ be all sequences of program statements in $\mathcal{P}$ that put the FSM in Figure 6, say $\mathbb{F}$, in its final state. As defined earlier, $T \subset \Sigma^*$ is the set of all feasible traces in the program, in this case, $\mathcal{P}$. If $T \cap B = \phi$, then the final state of $\mathbb{F}$ is never reached. Since B

and T are arbitrary languages and T is uncomputable, deciding if $T \cap B = \phi$ is an undecidable problem. Hence PDMC restricts the form of B and T by modeling B to be a regular language accepted by $\mathbb{F}$ ($B = L(\mathbb{F})$), and T as a context-free language accepted by a PDA $\mathbb{P}$ (of program $\mathcal{P}$). In general, we have $T \subseteq L(\mathbb{P})$, which then implies $T \cap B \subseteq L(\mathbb{F}) \cap L(\mathbb{P})$. Consequently, if $L(\mathbb{F}) \cap L(\mathbb{P})$ is empty, $T \cap B$ is definitely empty. However, if $L(\mathbb{F}) \cap L(\mathbb{P})$ is not empty, $T \cap B$ could either be empty or not. Since $L(\mathbb{F})$ is a regular language and $L(\mathbb{P})$ is a context-free language, $L(\mathbb{F}) \cap L(\mathbb{P})$ can be captured by a PDA, say **P**, and hence the final state of $\mathbb{F}$ is reached if and only if the PDA **P** accepts the empty language. There are efficient algorithms to determine if the language accepted by the PDA is empty [13]. Once **P** is constructed, PDMC checks to see if any final configuration is reachable in **P**. Chen and Wagner [7] use the preceding analysis to adapt PDMC for light-weight property checking. We use the preceding analysis for static trace generation. We call the FSMs such as the one used in Figure 6 as Triggers. By using Triggers, we have achieved two purposes:

- We have produced T_{ex}, the set of traces in the program that begin with e and end with x instead of $T' \subseteq T$. However, some traces can still be infeasible in the program because of data-flow insensitivity.

- The knowledge of $\mathcal{A} = \{a_1, a_2, a_3, ..., a_k\}$ allows us to extract $A(t)$ from any $t \in T_{ex}$.

3.4.3 Soundness

The consequence of using a context-free language for T introduces imprecision but retains the soundness of analysis. Infeasible traces might occur (being incomplete) because of data-flow insensitivity of the PDMC process, but all the program traces that put the FSM in its final state are reported (being sound). Since determining if $T \cap B = \phi$ is undecidable, no tool can be sound and complete at the same time. Consequently, there could be some infeasible API sequences in the database being fed to the FCPO miner. For example, in Figure 1, the variable i might never assume value 1; therefore, the trace a $\rightarrow$ f $\rightarrow$ e $\rightarrow$ c is infeasible in the program. Also, along some feasible paths, the implicit API ordering rules might be violated and APIs could be used incorrectly (producing buggy traces with actual errors). Hence the API sequence database might contain certain wrong API sequences. However, we assume that most programs that we analyze are well written. Hence, we expect only few feasible paths to be buggy, if at all. We expect to handle infeasible and buggy traces by selecting appropriate min_sup value. The traces generated by PDMC with Triggers can still be of infinite size (for example if there is a loop). We address this problem in Section 4.

To simplify the definitions, we had assumed in Section 3.3 that the APIs are empty methods, do not take any arguments, and return void. If we relax this assumption, then we should also consider those statements in the program that affect and are affected by a given API, say a. For example, if we have i = a(j); if (i != NULL) b(i);, then the program statements if (i != NULL) and b(i) are related to a. We should include such statements in the API sequence database before being fed to the FCPO miner. A possible API specification (a path in the partial order) could look like "the return value of a should always be compared to NULL before being passed to b". In Section 4, we implement simple data flow extensions to the PDMC process to include a few such statements in the API sequence database. We do not implement a potentially expensive pointer or alias analysis to consider all such statements. We intend to explore data-flow-sensitive model-checkers in future work for trace generation.

3.5 Scenario Extraction

A single static trace from the model checker might involve several API usage scenarios, being often interspersed. We have to separate different usage scenarios from a given trace, so that each scenario can be fed separately to the miner. A naive algorithm for scenario extraction would be to remove all duplicate APIs in a given trace and feed the resulting API sequence as a single scenario to the miner. But most traces have multiple scenarios around the same set of APIs. Furthermore, these different scenarios represent different usage patterns among the API set. The naive algorithm of deleting duplicates lead to loss of API ordering information and a drastic decrease in the number of scenarios fed to the miner.

We develop a refined scenario extraction algorithm is based on identifying producer-consumer chains (PC-chains) among APIs in the trace. The algorithm (henceforth called as the *PC-Chain* algorithm) is based on the assumption that related APIs have some form of data dependencies between them such as a producer-consumer relationship. In short, the PC-Chain algorithm first identifies PC-chains among APIs in traces and outputs them as scenarios. Isolated partial orders are then constructed among APIs in related PC-chains. Finally, partial orders are computed between heads of PC-chains, and these partial orders form partial order clusters. As an example, consider three sets of APIs (a, b, c, d), (e, f, g), and (h, i, j). The first API in each set produces a value that is consumed by the remaining APIs in the set. Figure 7(a) shows two traces produced by the model checker. The APIs are all interspersed and there are three scenarios in each trace. The arrows in Figure 7(a) show the PC-chains among related APIs. Figure 7(b) shows six different scenarios extracted from two traces. Figure 7(c) summarizes the six different API scenarios compactly as three isolated partial orders. Finally, Figure 7(d) merges the isolated partial orders into one big partial order. Algorithm 1 summarizes the PC-Chain scenario extraction algorithm.

Input: static traces
Output: partial order clusters
Identify all producers;
foreach *producer* **do**
    Identify consumers;
    Construct PC-chain;
    Output PC-chain as a scenario;
end
Construct isolated partial orders from scenarios;
Collect the head APIs from isolated partial orders;
Construct partial order among head APIs to form partial order clusters;
return patial order clusters;
  Algorithm 1: The **PC-Chain** algorithm for scenario extraction

3.6 Mining FCPOs

To mine the complete set of FCPOs, FRECPO [19] searches a set enumeration tree of transitive reductions of partial orders in a depth-first manner. In principle, a partial order can be uniquely represented as the set of edges in its transitive reduction. Moreover, all edges in a set can be sorted in the dictionary order and they can be written as a list. Therefore, we can enumerate all partial orders in the dictionary order. A set enumeration tree of partial orders can be formed: for orders R_1 and R_2, R_1 is an ancestor of R_2, and R_2 is a descendant of R_1 in the tree if and only if the list of edges in R_1 is a prefix of the list of edges in R_2. By a depth-first search of the set enumeration tree of transitive reductions of partial orders, FRECPO does not miss any frequent partial order.

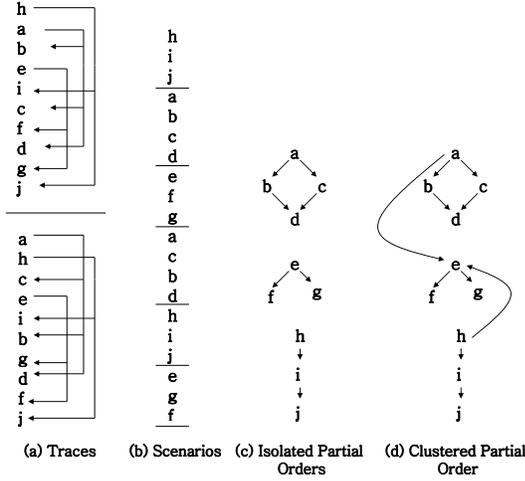To be efficient and scalable, FRECPO prunes the futile branches

(a) Traces   (b) Scenarios   (c) Isolated Partial Orders   (d) Clustered Partial Order

Figure 7: Scenario extraction example

and narrows the search space as much as possible. Basically, three types of techniques are used. First, FRECPO prunes infrequent items, edges, and partial orders. If a partial order R in the set enumeration tree is infrequent, then the partial orders in the subtree rooted at R, which are stronger than R, cannot be frequent. The subtree can be pruned. Second, FRECPO prunes forbidden edges. Not every edge can appear in the transitive reduction of a partial order. For example, if every string containing ac also contains ab and bc, then edge ac should not appear in the transitive reduction of any frequent closed partial order. Edge ac is called a forbidden edge. Removing the forbidden edges can also reduce the search space. Finally, FRECPO extracts transitive reductions of frequent partial orders directly and does not need to compute the transitive reductions.

3.7 Specification Extraction

One of the major problems with specification miners is the number of false specifications that they produce. False specifications lead to false positives, which drastically decrease the programmer productivity. Our tool learns frequent partial orders from program traces. But frequent partial orders might just be usage patterns specific to the analyzed client code, and not a universal specification. Hence we need to separate real specifications from frequent usage patterns that are only specific to the analyzed client code. We propose an algorithm to address this issue (henceforth called as the *Mine-Verify* algorithm). The Mine-Verify algorithm uses two disjoint sets (randomly split) of clients using the same set of APIs. The first set is used for mining specifications (henceforth, we call this set of clients as *mine clients*). The specifications mined by our framework, by analyzing mine clients, are verified against another set of clients (we call this set of clients as *verify clients*) using a static compile-time model checker. This process of mining specifications from one set of client programs (mine clients) and verifying it against another set (verify clients) enables our tool to reduce the number of mined false specifications. With a high probability, a real specification mined from mine clients will either be satisfied by verify clients (specification compliance) or violated a few times (bugs). This mechanism is based on an assumption [4, 8, 27] that a client generally uses an API or a set of APIs correctly most of the times. On the other hand, if a specification mined from mine clients causes too many violations in verify clients, we flag the mined specification as a possible frequent usage pattern; frequent

usage patterns are not universal API specifications. We illustrate the Mine-Verify algorithm in Section 5. Algorithm 2 summarizes the Mine-Verify algorithm.

Input: Two disjoint (randomly split) set of clients, **mine clients** and **verify clients**, user-specified APIs
Output: Potential Real specifications
Mine specifications from **mine clients**;
Let $S = (S_1, S_2, ...S_n)$ be the specifications mined;
$i = 1$;
T = user specified threshold (a small number);
while $i <= n$ **do**
    Check S_i against **verify clients**;
    Let V = Number of violations;
    if $V == 0$ **then**
      | Flag S_i as a real specification;
    end
    else if $V > 0$ *AND* $V <= T$ **then**
      | Flag S_i as a real specification;
      | Potential bugs found;
    end
    else if $V > T$ **then**
      | Flag S_i as NOT a specification;
    end
    $i + +$;
end

Algorithm 2: The **Mine-Verify** algorithm for finding real specifications

3.8 Complexity

PDMC constructs PDA $\mathbb{P}$ from the program Control Flow Graph (a directed graph $G = (N, E)$) where each node represents a program point and each edge represents a valid program statement. PDMC takes $O(E)$ time to construct the PDA $\mathbb{P}$ from the CFG G, takes $O(E \times |Q|)$ (Q is the number of states in the FSA) for computing $\mathbf{P}$, the product of FSA $\mathbb{F}$ and PDA $\mathbb{P}$, takes $O(|Q|^2 \times E)$ for deciding if the PDA $\mathbf{P}$ is empty and $O(|Q|^2) \times lg|Q| \times E \times lgN$ for backtracking. The derivations are shown by Chen [6].

It has been shown that the problem of counting the complete set of frequent closed partial order is #P-complete. In other words, FRECPO is of exponential complexity with respect to $|\mathcal{A}|$. However, FRECPO is pseudo-linear. That is, the runtime is linear with respect to the number of frequent closed partial orders in the data set. In practice, the number of significant frequent closed partial orders of APIs for a program is often small. Thus, it is highly feasible and effective to use FRECPO in our application context.

4. IMPLEMENTATION

To generate static traces, we adapted a publicly available model checker called MOPS [7]. To mine FCPOs, we adopted an FCPO miner called FRECPO [19]. The process of generating error traces from a final configuration $\langle q, \omega \rangle$ (ω is the stack content containing a list of return addresses) of PDA $\mathbf{P}$ is called *backtracking* [6]. With the knowledge of a user-specified set of APIs, $\mathcal{A} = \{a_1, a_2, a_3, ..., a_k\}$, our framework extracts $A(t)$ from any trace t output by MOPS. The PDMC process outputs a graph in which certain paths map to violation paths in the program [6]. Multiple program paths (and hence graph paths) can violate a given property specified by a FSM (such as the one shown in Figure 5), and many such violations could be similar because they indicate the same programming error. So instead of reporting all program traces that violate a given property, the MOPS model checker clusters similar traces and reports the shortest trace as a candidate trace for each violation. This mechanism would save the user's time considerably

Table 1: X11 client programs used in our evaluation

appres	beforelight	bitmap	dpsexec	dpsinfo	editres	glxgears	glxinfo
iceauth	ico	listres	luit	makepsres	oclock	proxymngr	rstart
setxkbmap	showfont	smproxy	texteroids	twm	viewres	x11perf	xauth
xbiff	xcalc	xclipboard	xclock	xcmsdb	xconsole	xditview	xdpyinfo
xev	xeyes	xf86dga	xfd	xfindproxy	xfontsel	xfsinfo	xfwp
xgamma	xgc	xhost	xinit	xkbevd	xkbprint	xkbutils	xkill
xload	xlogo	xlsatoms	xlsclients	xlsfonts	xmag	xman	xmessage
xmh	xmodmap	xpr	xrandr	xrdb	xrefresh	xset	xsetmode
xsetpointer	xsetroot	xstdcmap	xterm	xtrap	xvidtune	xvinfo	xwud

Table 2: Call frequencies for selected X11 APIs

XCreateGC	56
XOpenDisplay	32
XCreateWindow	26
XOpenIM	4
XQueryBestSize	2
XFreeModifiermap	1

Figure 8: Trigger used to generate functions, APIs, and expressions related to `XOpenIM`

because the user has to review each trace manually. However, for our purposes, given a Trigger, we need all the traces in the program that contain the APIs specified in the Trigger. We modified the backtracking algorithm of MOPS, wherein, instead of clustering traces, we consider all program paths that satisfy the Trigger, and output a random number of traces by random walking the graph generated by the PDMC process. In our experiments, we specified a threshold (20 in our experiments) for the number of traces to be generated from each mine client.

Because the basic MOPS static checker is data-flow insensitive, it assumes that a given variable might take any value. Therefore, it assumes that both branches of a conditional statement may be taken and that a loop may execute anywhere between zero to infinite iterations. Data-flow insensitivity causes MOPS to output infeasible traces. Furthermore, the trace size and the number of traces can be infinite due to loops. MOPS monitors backtracking and aborts if it detects a loop. We write extensions to the MOPS pattern matching [7]; these extensions make it possible to track the value of variables that take the return status of an API call along the different branches of conditional constructs. For each possible execution sequence, our extensions associate a value to the variable that is being tracked using pattern matching. MOPS pattern matching allows our framework to correlate two program statements related by program variables (as an example, `FILE* fp = fopen(...)` and `fread(fp)` are related through the file pointer variable, `fp`). Our extensions enable our framework to mine properties such as "If API `a` returns `NULL`, then API `b` should always be called along the `NULL` path". Our current implementation does not consider aliasing. Hence our framework might miss some traces in which the API return variables are aliased.

5. EVALUATION

We applied our framework on 72 client programs from the `X11R6.9.0` distribution. The analyzed client programs use APIs for the X11 windowing system, with roughly 200K LOC in total. Table 1 lists the X11 client programs used in our evaluation. We selected X11 client programs because the Inter-Client Communication Conventions Manual (ICCCM) [21] from the X Consortium standard describes several rules for how well-behaved programs should use the X11 APIs, serving as an oracle. For each experiment, we randomly choose 36 clients as mine clients. The remaining 36 clients are verify clients.

Roughly, 700 distinct X11 APIs were used across 72 client programs. For each X11 API, we calculated the number of call sites across all client programs. Table 2 shows the call site frequencies
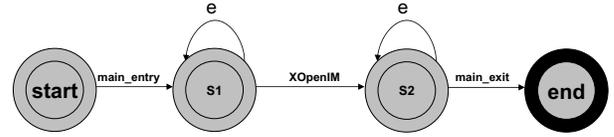
for selected X11 APIs. APIs such as `XCreateGC`, `XOpenDisplay`, and `XCreateWindow` were called quite often (more than 25 times) among 72 clients. But the usage of APIs such as `XOpenIM`, `XQueryBestSize`, and `XFreeModifiermap` were relatively sparse (less than 5 times) across all the clients. We next present an illustrative example of how specifications are mined by our tool around `XOpenIM` API. This simple example demonstrates how our tool selects specifications from frequent usage patterns using the Mine-Verify algorithm. The example also demonstrates how our tool handles infeasible traces that appear because of the absence of deep data flow analysis in our model checker.

First, we randomly partition the X11 clients into mine clients and verify clients. Then we use our tool to generate static traces from each mine-client program using the Trigger shown in Figure 8. For the purposes of this illustration and clarity, we pick $t = 20$ distinct traces from the resulting trace set. States $S1$ and $S2$ in the Trigger have self transitions on e. e is any function/API invoked before/after `XOpenIM` or any expression that uses or defines the return variable and input parameters of `XOpenIM`. The Trigger causes the model checker to output paths (as traces from `main_entry` to `main_exit`) that involve a call to `XOpenIM`. The traces contain function calls and APIs that are ancestors or descendants of `XOpenIM` in the program call graphs of each client that `XOpenIM` appears in. The model checker also outputs expressions (in the program) that share a *use-def* or *def-use* relation (we do not consider aliasing) with `XOpenIM`'s return variable or input parameters (again, across all clients with a call site to `XOpenIM`). These traces are then fed to the partial order miner after scenarios are extracted. The miner summarizes the traces seen from different program call graphs (from different mine clients) as a single partial order (for a given minimum support value, m). Our tool produces all partial orders with support s greater than or equal to m. Our tool then picks partial orders with a higher support value as likely specifications. However, a certain frequent usage pattern (not a specification) specific to the mine clients might be output as a specification. We next show how the Mine-Verify algorithm can be used to separate real specifications from false one.

We parsed the traces generated from the Trigger shown in Figure 8, and picked 5 APIs, including `XOpenIM`, which frequently appear in all traces. These APIs were `XtDisplay` (0), `XOpenIM`(1), `XGetIMValues` (2), `XFree` (3), and `XCloseIM` (4). We use number symbols (in braces) to represent the APIs and we use $< xy >$ to indicate that API y follows API x. In the extracted partial orders, $<$

Table 3: Statistics for the specifications mined by our tool

Name	Seed	Mined	Related APIs	Extra Specs	MV filtered	Real	FALSE	Missed
PrsTransTbl	XtParseTranslationTable	Yes	2	0	0	0	0	0
XPutImage	XPutImage	No	5	8	2	4	2	1
XSaveContext	XSaveContext	Yes	3	3	0	3	0	0
XSetFont	XSetFont	Yes	2	2	0	2	0	0

$14 >$ (`XCloseIM` follows `XOpenIM`) and $< 23 >$ (`XFree` follows `XGetIMValues`) had a support of 8 each, $< 01 >$ (`XOpenIM` follows `XtDisplay`) had a support of 20, and $< 12 >$ (`XGetIMValues` follow `XOpenIM`) had a support of 16. Low support of 8 for $< 14 >$ was due to the fact that if `XOpenIM` fails (returning `NULL`), the enclosing function returns without calling `XCloseIM`. The case for `XGetIMValues` was similar except that `XOpenIM` returns `NULL` on failure, while `XGetIMValues` return `NULL` on success. The inclusion of traces in which APIs fail reduced the support for $< 14 >$ and $< 23 >$. However, using the tool developed in our previous work [1], we can mine interface details (such as return values on failure) for a given API automatically from the source code. The mined information can be used to avoid traces in which the APIs fail (by implementing simple data flow extensions to the model checker). When this mechanism was used, the support for $< 14 >$ and $< 23 >$ increased to 18. With the minimum support value of 16/20, the likely specifications were $< 01 >$, $< 12 >$, $< 14 >$, and $< 23 >$. We verified these likely properties against verify clients. $< 14 >$ gave a single violation. This pattern is a specification, leading to a bug. $< 23 >$ gave no violation. This pattern is a specification finding no bug. $< 01 >$ gave a very large number of violations (we stopped after 10 violations were reported on verify clients). This pattern is a false specification, which is rejected by our tool using the Mine-Verify algorithm. Finally, $< 12 >$ gave no violation with verify clients. This pattern is a false specification uncaught by our tool. This false specification is not caught by our tool because of the limited ways in which `XOpenIM` was used in our subjects. In both mine clients and verify clients, all instances of `XOpenIM` were followed by `XGetIMValues`, though it is not necessary. Other APIs such as `XSetIMValues` can also follow `XOpenIM`. We consulted the ICCCM manual to confirm our results.

We manually inspected each trace produced by our tool against the source code. Of the 20 traces, only one trace was found to be infeasible. The trace entered the conditional `if(!found)`, even when the variable `found` was `true`. Infeasible traces appear because of the data-flow-insensitivity of our model checker. We expect to handle infeasible traces by specifying minimum support values appropriately in our experiments. We next demonstrate how our tool can be used to mine specifications around those mined by Ammons et al. [5] (for convenience, we call their approach dynamic-trace miner). We used a minimum support value of 0.8 in our experiments. A mined pattern with more than 3 violations in verify clients was flagged as a potential usage pattern and was not considered a specification.

Table 3 summarizes our results. An implementation limitation of our model checker is that it cannot handle function pointers or callbacks. Hence, we considered only those X11 specifications mined by the dynamic-trace miner that do not involve callbacks. The four specifications from dynamic-trace-miner used in our evaluation are shown in Column 1 of Table 3. Our first goal was to specify an API (we call this API the seed API) from the specification mined by the dynamic-trace miner and check if our tool can mine the same specification. Column 2 shows the seed API given to our tool. We could mine all specifications mined by dynamic-trace miner except for one (*XPutImage*) as Column 3 shows. *XPutImage* requires that the image and graphics context passed to `XPutImage`

Table 4: Statistics for the specifications mined around `XOpenDisplay`

Specifications mined	10
MV filtered	4
Real	5
FALSE	1
Missed	1

API must have been created on the same display. The data-flow analysis required to mine this specification was not present in the model checker that we used and hence we missed this specification.

The second goal was to mine specifications among APIs related to the specified seed, not mined by the dynamic-trace miner. Related APIs are those APIs that have simple data dependencies with the seed API. Related APIs either produce a value that the seed API consumes or consume a value produced by the seed API. Our approach first gathers the APIs related to the seed API. Column 4 displays number of APIs that are determined by our tool to be related to the seed API. Then our tool mines frequent usage scenarios as partial orders and produces ones with high support as likely specifications. Specifications learnt around the related API `XOpenDisplay` are shown separately later this section because a very large number of APIs interact with `XOpenDisplay`. Column 5 shows the number of likely specifications mined by our tool, around the specification mined by the dynamic-trace miner. After running the Mine-Verify algorithm on the specifications mined by our tool (Column 6), we separate out true specifications (Column 7) from false specifications (Column 8). As we had prior knowledge of specifications from the ICCCM manual, we noted that our tool missed a specification involving `XCreateImage`, a related API, owing to data-flow-insensitivity of our model checker (Column 9).

We next present our results for specifications learnt around the `XOpenDisplay` API. `XOpenDisplay` returns a pointer to the *Display* structure that serves as the connection to the X server and that contains all the information about that X server. `XOpenDisplay` connects the application to the X server through TCP or DEC-net communications protocols, or through some local inter-process communication protocol. The pointer returned by `XOpenDisplay` is consumed by a large number of X11 APIs, scattered across procedure boundaries. Hence `XOpenDisplay` makes an interesting case study for our tool. Figure 9 shows a partial order learnt by our tool, being a frequent usage scenario. Table 4 shows the statistics for the specifications learnt by our tool around `XOpenDisplay`.

6. RELATED WORK

Various mining approaches have been developed to extract API usage rules or patterns out of source code or execution traces. From source code, CodeWeb developed by Michail [18] mines association rules [2] such as that application classes inheriting from a particular library class often instantiate another class or one of its descendants. PR-Miner developed by Li and Zhou [15] uses frequent item set mining [10] to extract implicit programming rules from source code and detect their violations for detecting bugs. The tool developed by Williams and Hollingsworth [25] and DynaMine developed by Livshits and Zimmermann [17] mine simple
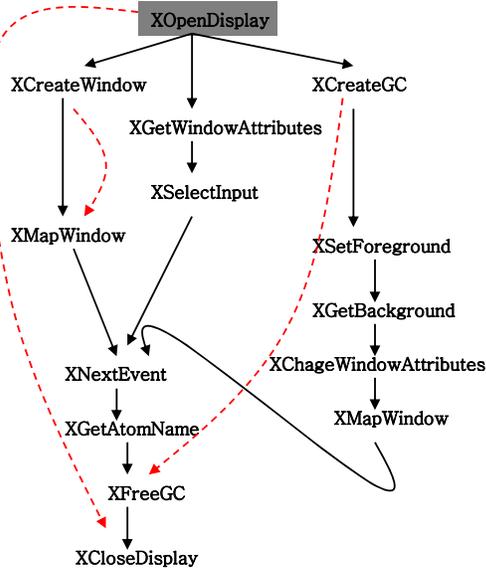
Figure 9: A usage scenario around `XOpenDisplay` API as a partial order. Lower support values produce larger partial orders. Higher support values produce smaller partial orders (specifications). Specifications are shown with dotted lines.

rules from software revision histories. These rules involve mostly method pairs. From error-handling code, Weimer and Necula [23] mine temporal safety rules that involve pairs of API calls. From source code, Engler et al. [8] also infer API rules that involve pairs of API calls. Perracotta developed by Yang et al. [27] infers temporal properties (in the form of pre-defined templates involving two API calls) from program executions. From source code, MAPO developed by Xie and Pei [26] mines API usage patterns in the form of frequent subsequences [22]. Different from the API rules or patterns mined by these previous approaches, our framework mines more complicated API usage patterns in the form of partial orders from source code. The mined partial orders provide important, useful API ordering information that is not provided by patterns mined by previous approaches. Different from most of the preceding approaches using intra-procedural static program information, our approach considers the program control flow and hence can mine patterns from APIs being spread across procedural boundaries. In our evaluation, we observed that most X11 APIs are spread across procedural boundaries.

LtRules developed by Liu et al. [16] receives a given set of APIs, creates all possible API usage orders (as FSMs) determined by a set of pre-defined templates (involving with one or two API calls), and checks the generated API FSMs against "good" clients by using the BLAST [11] model checker. The template instantiations that pass the BLAST test are considered to be specifications. These specifications are used for verifying compliance in other clients. This technique requires "good" reference test programs and fails to infer properties if reference programs have bugs. In our approach, we make no assumptions about mine clients and verify clients except that they are disjoint and randomly split. Our framework generates Triggers from user-specified APIs instead of concrete properties like LtRules. As we use mining techniques for pattern inference, we can still infer properties from buggy programs. In addition, our mined partial order patterns are beyond the mined patterns specified by their user-defined, simple property templates.

Kremenek et al. [14] incorporate many disparate sources of ev-

idence and use factor graphs to learn specifications around functions/APIs called very few times in the program. Our approach learns from multiple static paths (each leading to a trace) that go through API call sites and works for APIs with very few call sites. Our tool also outputs APIs and expressions related to the specified API and learns usage scenarios and specifications among them.

Ammons et al. [4,5] infer API usage properties by observing program execution and concisely summarizing the frequent interaction patterns as probabilistic finite state automaton (PFSA) based on a PFSA learner [20]. Their approach requires setup of runtime environments and availability of sufficient system tests that exercise various parts of the program. Our framework requires only system source code. In addition, the PFSA learner [20] used by their approach cannot infer frequent API usage patterns from traces where other APIs not related to the patterns also occur. Our tool gathers APIs related to the specified API and learns usage scenarios and specifications among them. In our evaluation, we showed how our approach can mine specifications around those mined by their approach.

A number of approaches [3, 12, 24] have been developed to apply static analysis or model checking on the API implementation code to synthesize permissive API usage patterns that are allowed by the API implementation. Different from these approaches, our framework analyzes API client code (rather than API implementation code) and applies a miner on static traces extracted from the client code. In general, the API usage partial orders mined from API client code are a subset of permissive API usage patterns allowed by API implementation code. Our mined API usage patterns may provide more guidance (with a focused set of commonly observed API usage patterns) to the programmers in reusing APIs.

7. DISCUSSION AND FUTURE WORK

The model checker used in our framework is data-flow-insensitive. This limitation leads to infeasible traces. In our experiments, we handled infeasible traces by specifying support values appropriately. However, a large number of infeasible traces might lead to false specifications that might not be caught by the Mine-Verify algorithm. In our experiments, the number of generated traces were in hundreds (both intra- and inter-procedural) and it was not worthwhile to manually inspect each trace against the source code to determine whether the trace was feasible or not. Hence we have not quantified the actual number of infeasible traces in our evaluation. Instead, in our evaluation, we measured the number of real and false specifications mined by our tool, using the ICCCM manual as oracle. In future work, we plan to explore the utility of data-flow-sensitive model checkers such as BLAST [11] for trace generation. However, a downside of using data-flow-sensitive model checkers for trace generation is that they are generally not scalable. The data-flow-insensitive model checker adapted in our experiments, MOPS, is scalable and was used to verify code bases as large as the entire Linux distribution (30M LOC). The model checker that we used does not handle function pointers and callbacks. Hence we could not mine X11 specifications that involve callbacks. Finally, the FCPO miner employed in our framework does not handle duplicate strings. We handled this implementation limitation by appropriately modifying the scenario extraction algorithm. In future work, we plan to apply PFSA (Probabilistic Finite State Automata) learners [20] on the static traces generated by the model checker. As PFSAs allow loops unlike partial orders, we expect to mine interesting classes of properties beyond those mined in our experiments. Although we have applied our framework on clients written in C, the basic idea is applicable to even object-oriented languages such as Java and C#.

8. CONCLUSIONS

Usage patterns and ordering rules among APIs are often not documented by API developers. In this paper, we presented a framework to automatically extract usage scenarios among user-specified APIs as partial orders, directly from API client code. Specifications were extracted from the frequent partial orders using our Mine-Verify specification extraction algorithm. We adapted a compile-time model checker to generate interprocedural control-flow-sensitive static traces related to the APIs. From the static traces, our scenario extraction algorithm extracted different API usage scenarios, which were fed to a miner. The miner summarized different usage scenarios as compact partial orders, from which specifications were extracted. We applied our framework on 72 X11 clients with 200K LOC in total and compared our approach with an existing specification miner. Our results highlighted the unique benefits of our approach and showed that the extracted API partial orders are useful in assisting effective API reuse and checking.

9. REFERENCES

[1] M. Acharya, T. Xie, and J. Xu. Mining interface specifications for generating checkable robustness properties. In *Proc. International Symposium on Software Reliability Engineering (ISSRE)*, pages 311–320, 2006.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[3] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 98–109, 2005.

[4] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 4–16, 2002.

[5] G. Ammons, D. Mandein, R. Bodik, and J. Larus. Debugging temporal specifications with concept analysis. In *Proc. Conference on Programming Language Design and Implementation (PLDI)*, pages 182–195, 2003.

[6] H. Chen. *Lightweight Model Checking for Improving Software Security*. PhD thesis, University of California, Berkeley, 2004.

[7] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, 2002.

[8] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72, 2001.

[9] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking push down systems. In *Proc. International Conference on Computer Aided Verification (CAV)*, pages 232–247, 2000.

[10] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proc. IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

[11] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. Workshop on Model Checking Software*, pages 235–239, 2003.

[12] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proc. European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 31–40, 2005.

[13] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[14] T. Kremenek, P. Twohey, G. Back, D. Engler, and A. Ng. From uncertainty to belief: Inferring the specification within. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pages 161–176, 2006.

[15] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 306–315, 2005.

[16] C. Liu, E. Ye, and D. J. Richardson. Software library usage pattern extraction using a software model checker. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 301–304, 2006.

[17] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proc. European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 296–305, 2005.

[18] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proc. International Conference on Software Engineering (ICSE)*, pages 167–176, 2000.

[19] J. Pei, H. Wang, J. Liu, K. Wang, J. Wang, and P. Yu. Discovering frequent closed partial orders from strings. *IEEE Transactions on Knowledge and Data Engineering*, 18(11):1467–1481, 2006.

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

[21] D. Rosenthal. *Inter-client communication Conventions Manual (ICCCM), Version 2.0. X Consortium, Inc.* 1994.

[22] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. International Conference on Data Engineering (ICDE)*, pages 79–90, 2004.

[23] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476, 2005.

[24] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 218–228, 2002.

[25] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480, 2005.

[26] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proc. International Workshop on Mining Software Repositories (MSR)*, pages 54–57, 2006.

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