

Finding Related Events for Specification Mining

Ziying Dai*, Xiaoguang Mao*^{†‡}, Liqian Chen*, Yan Lei*, Yi Zhang[§]

*College of Computer, National University of Defense Technology, Changsha 410073, China

*Laboratory of Science and Technology on Integrated Logistics Support,
National University of Defense Technology, Changsha 410073, China

[§]Naval Academy of Armament, Beijing 100036, China

[‡]Corresponding author, email: xgmao@nudt.edu.cn

Index Terms—Specification mining, Static analysis, Component interfaces, Related events

I. INTRODUCTION

In contemporary software development practice, programmers reuse components by invoking their APIs to construct large systems. These APIs often involve constraints on the temporal order of method calls. For the example of the file usage, a programmer should first *open* a file, then *read* and/or *write* its content, and at last *close* it. Trying to read a closed file will cause exceptions to be thrown. Such constraints are often represented as a finite state machine (FSM) with a set of related events (typically method calls) as its alphabet. A set of events are *related* if interactions among them possibly obey some meaningful temporal specifications. In recent years, various specification mining techniques have been developed to automatically mine API specifications from API client programs [1]. A typical API specification miner conceptually has three steps. First, it decide which events are related. Second, different interactions among related events (which are sub-traces and sample strings of the specification FSM) are extracted either from source code of client programs or from their execution traces. Third, extracted interactions are passed to customized or off-the-shelf FSM learners which generalize these sample sub-traces to recover the specification FSM.

Finding related events is very important for the API specification mining. On one hand, if not all related events are considered together, the corresponding specification cannot be mined. On the other hand, because specification miners are limited to the set of behaviors (observed behaviors for dynamic miners) of client programs, grouping unrelated events together will cause more mining cost (the typical PFSA learner [2] has the cubic time complexity to the length of the trace) and produce specifications with poor *recall* (that is the percentage of correct behaviors that have been mined). This problem is more compelling when to mine API specifications of multiple interacting objects. Multi-object specifications are important and more expressive than single-object ones in that they can capture constraints on methods from different objects. The challenge of mining API specifications of multiple objects lies in the fact that interactions among different objects are common and often complex.

In this paper, we propose a static analysis approach to find related methods of object-oriented components from their

source code. Then a set of events are related if they are calls of related methods. We first search *critical predicates* which a statement is control-dependent on to throw an exception. These predicates are critical because their values can detect errors that are typically signaled by throwing exceptions in object-oriented programs. Then, for each critical predicate, we find all methods that have this critical predicate or mutate a variable defining it. These methods relate to each other on the rationale that unless they access shared variables, their relative ordering is independent of each other.

II. RELATED WORK

Our approach is semantic-based and can find related events from multiple interacting objects. It does not directly reason about critical predicates and they can be arbitrarily complex. Many existing miners focus on single object specifications [3][4], which typically consider all events of an object related [3]. Whaley et al. [4] further divide events of an object into small groups which consist of calls of methods that refer to the same field of the object. Other multi-object API specification miners utilize various heuristics to solve this problem. Various syntactic scopes are employed such as that related events are within the execution of a method of the client program [5], within the execution of a unit test of the target API [2], within the code of a method of the client program [6], and within a package [2][5]. Lee et al. [2] further confine related events to those that share at least one common argument (parameters, receivers and returns). These approaches often groups unrelated events together due to the coarse syntactic scopes and can omit semantically related events due to the poor quality of used syntactic artifacts (e.g. unit tests [2]).

III. APPROACH TO FIND RELATED EVENTS

We discuss our approach for Java components. It conceptually consists of four phases as illustrated in Figure 1. The input is the source code of target components. The output are sets of related events. Existing specification miners can directly use the output to mine better specifications. First, we perform a pointer and side effect analysis [8]. The pointer analysis provides aliases information for downstream analysis. For each method m , the side effect analysis computes the set W_m of global variables that m mutates. Global variables of m include m 's parameters (including the receiver *this*), m 's return object if it is created by m , and fields reachable

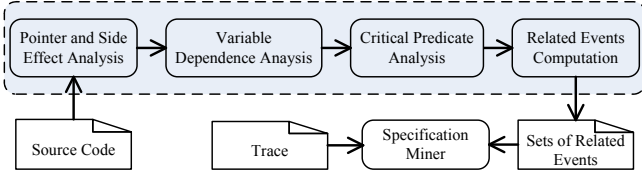


Fig. 1. Overview of our approach (the highlighted part) and its integration with existing specification miners.

from them. The analysis propagates mutation effects interprocedurally by mapping parameters of the callee to objects in the calling context. For the code in Figure 2, $W_{iterator} = \{return.expectedModCount\}$ with $return$ denoting the returned object, and $W_{add} = W_{AbstractList.(init)} = \{this.modCount\}$.

In the second phase, we perform the *variable dependence* analysis that determines which global variables are dependent on by the value of each variable at every program point of a method m . Our analysis forwardly propagates information along the control flow edges. At control flow join points, we merge the incoming sets of global variables for each variable. For an assignment statement “ $v = E$ ”, the set of global variables dependent on by v is updated to the set of all global variables dependent on by a variable appearing in E . We say that this set of global variables, denoted as $Dep(E)$, is dependent on by E . Other statements do not change the analysis result. We compute a summary Sum_m of m by unionizing each set at each exit point of m for every variable. If m has return values, we also compute a set of global variables RD_m dependent on by the return values. For each statement “return E ”, we determine the predicate P on which this statement is control-dependent by constructing the program dependence graph [7] of m . Then RD_m includes all global variables dependent on by E and P . If m has multiple return statements, the result is the union of the sets for each return statement. The analysis processes each method only once and starts from the leaves of the call graph. To propagate information interprocedurally and use return values of a called method m , we use Sum_m and RD_m by mapping parameters of the callee to objects in the calling context. For example, $Dep(modCount \neq expectedModCount) = \{this.this\$0.modCount, this.expectedModCount\}$ with the field $this\$0$ denoting the outer object of the `Iterator` object (at the line 12 in Figure 2).

In the third phase, we find *critical predicates* P_m for each method m . A critical predicate controls whether an escaped exception can be thrown. An exception *escapes* if it is not caught within the body of m . We first construct the program dependence graph G_m [7] for m . Then, P_m includes every predicate in G_m that a statement is control-dependent on to throw an escaped exception. For the example in Figure 2, $P_{next} = P_{remove} = P_{checkForComodification} = \{modCount \neq expectedModCount\}$. For a method call inside m , we add all of the critical predicates of the called method to P_m except for those whose guarded exception cannot escape from m . Critical predicates here are not path conditions based on the observations that (1) path conditions are often method-specific; and (2) a critical state is often detected through a specific

```

1 public abstract class AbstractList<E> {
2   protected transient int modCount = 0;
3   public Iterator<E> iterator() { return new Itr(); }
4   public boolean add(E e) { ...; modCount++; ... }
5   private class Itr implements Iterator<E> {
6     int expectedModCount = modCount;
7     public boolean hasNext() { return cursor != size(); }
8     public E next() { checkForComodification(); ... }
9     public void remove() { if (lastRet == -1) throw new IllegalStateException();
10    checkForComodification(); ... }
11    final void checkForComodification() {
12      if (modCount != expectedModCount)
13        throw new ConcurrentModificationException(); ... } ... }

```

Fig. 2. Code of the `java.util.AbstractList` and its inner class `Itr`.

predicate that can be reused among different methods.

The last phase is to compute related events. For each critical predicate p , the set of related methods R_p includes every method m with p as its critical predicate, that is, $p \in P_m$; In addition, R_p includes every method m that mutates one of the global variables control-dependent on by p , that is, $W_m \cap Dep(p) \neq \emptyset$. In object-oriented programs, it is typical to signal a state error of a component by throwing an exception. The predicate control-dependent on by an escaped exception is critical in that its value is used to detect errors including those caused by illegal sequences of method invocations. Methods mutating a global variable v that is control-dependent on by p are related because they can mutate v 's value so that the exception guarded by p can be thrown.

We convert a set of related methods corresponding to the critical predicate p to an event specification [2] by denoting the same root object of the global variables in $Dep(p)$ with a unique symbol and keeping the signatures of related methods attached with parameters and/or return symbols. For the example in Figure 2, the event specification for the critical predicate “ $modCount \neq expectedModCount$ ” is $\{\langle init \rangle(l), add(l), iterator(l, i), next(i), remove(i)\}$ with l denoting a `List` and i denoting its `Iterator`. The first parameter of a method denotes its receiver and the second if any denotes its return. None of existing approaches [2][5][6] can get this result. They mistakenly group the unrelated event `hasNext(i)`¹ into this set and omit the related event `remove(i)`.

REFERENCES

- [1] M. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, (99):1, 2012.
- [2] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *ICSE*, 2011, pp. 591–600.
- [3] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically generating test cases for specification mining. *IEEE Transactions on Software Engineering*, 38(2):243–257, 2012.
- [4] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, 2002, pp. 218–228.
- [5] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE*, 2009, pp. 371–382.
- [6] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE*, 2009, pp. 383–392.
- [7] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its uses in optimization. *ACM Transaction on Programming Languages and Systems*, 9(3):319–349, 1987.
- [8] A. Sălciuanu and M. Rinard. Purity and Side Effect Analysis for Java Programs. In *VMCAI*, 2005, pp. 199–215.

¹`hasNext(i)` is related to `next(i)` in a sense and they can be grouped together for another critical predicate “`this.cursor >= this.this$0.size`”.