Static Specification Mining Using Automata-Based Abstractions

Sharon Shoham
Technion Israel Institute of Technology
sharonsh@cs.technion.ac.il

Eran Yahav Stephen Fink Marco Pistoia
IBM T. J. Watson Research Center
{eyahav,sjfink,pistoia}@us.ibm.com

ABSTRACT

We present a novel approach to client-side mining of temporal API specifications based on static analysis. Specifically, we present an interprocedural analysis over a combined domain that abstracts both aliasing and event sequences for individual objects. The analysis uses a new family of automata-based abstractions to represent unbounded event sequences, designed to disambiguate distinct usage patterns and merge similar usage patterns. Additionally, our approach includes an algorithm that summarizes abstract traces based on automata clusters, and effectively rules out spurious behaviors.

We show experimental results mining specifications from a number of Java clients and APIs. The results indicate that effective static analysis for client-side mining requires fairly precise treatment of aliasing and abstract event sequences. Based on the results, we conclude that static client-side specification mining shows promise as a complement or alternative to dynamic approaches.

Categories and Subject Descriptors. D.2.4 [*Program Verification*]; D.2.1 [*Requirements/Specifications*]

General Terms. Algorithms, Verification, Abstract Interpretation

Keywords. Specification Mining, Static Analysis, Typestate

1. INTRODUCTION

There is only one thing more painful than learning from experience and that is not learning from experience.
– Archibald MacLeish

Specifications of program behavior play a central role in many software engineering technologies. In order to apply such technologies to software lacking formal specifications, much research has addressed *mining* specifications directly from code [1, 2, 5, 20, 21, 22, 13, 8, 12, 15, 7].

Most such research addresses *dynamic analysis*, inferring specifications from observed behavior of representative program runs. Dynamic approaches enjoy the significant virtue that they learn from behavior that definitively occurs in a run. On the flip side, dynamic approaches can learn *only* from available representative runs; incomplete coverage remains a fundamental limitation.

The Internet provides access to a huge body of representative clients for many APIs, through myriad public code repositories and search engines. However, the amount of code available for inspection vastly exceeds the amount of code amenable to automated dynamic analysis. Dynamic analysis requires someone to build, deploy, and set up an appropriate environment for a program run. These tasks, difficult and time-consuming for a human, lie far beyond the reach of today's automated technologies.

To avoid the difficulties of running a program, a tool can grab code, and apply static program analysis to approximate its behavior. For this reason, static analysis may add value as a complement or alternative to dynamic analysis for specification mining.

Static analyses for specification mining can be classified as *component-side*, *client-side*, or both. A component-side approach analyzes the implementation of an API, and uses error conditions in the library (such as throwing an exception) or user annotations to derive a specification.

In contrast, client-side approaches examine not the implementation of an API, but rather the ways client programs use that API. Thus, client-side approaches can infer specifications that represent how a particular set of clients uses a general API, rather than approximating safe behavior for all possible clients. In practice, this is a key distinction, since a specification of non-failing behaviors often drastically over-estimates the intended use cases.

This paper addresses static analysis for client-side mining, applied to API specifications for object-oriented libraries. The central challenge is to accurately track sequences that represent typical usage patterns of the API. In particular, the analysis must deal with three difficult issues:

- **Aliasing.** Objects from the target API may flow through complex heap-allocated data structures.
- **Unbounded Sequence Length.** The sequence of events for a particular object may grow to any length; the static analysis must rely on a sufficiently precise yet scalable finite abstraction of unbounded sequences.
- **Noise.** The analysis will inevitably infer some spurious usage patterns, due to either analysis imprecision or incorrect client programs. A tool must discard spurious patterns in order to output intuitive, intended specifications.

We present a two-phase approach consisting of (1) an *abstract-trace collection* to collect sets of possible behaviors in client programs, and (2) a *summarization* phase to filter out noise and spurious patterns. Specifically, the main contributions of this paper are:

- a framework for client-side specification mining based on flow-sensitive, context-sensitive abstract interpretation over a combined domain abstracting both aliasing and event sequences,
- a novel family of abstractions to represent unbounded event sequences,
- novel algorithms to summarize abstract traces based on automata clusters, and
- results from a prototype implementation that mines several interesting specifications from non-trivial Java programs.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

```
class SocketChannelClient {
 void example() {
  Collection<SocketChannel> channels = createChannels();
  for (SocketChannel sc : channels) {
   sc.connect(new InetSocketAddress("tinyurl.com/23qct8",80));
   while (!sc.finishConnect()) {
    // ... wait for connection ...
   }
   if (?) {
    receive(sc);
   } else {
    send(sc);
   }
  }
  closeAll(channels);
 }
 void closeAll(Collection<SocketChannel> chnls) {
   for (SocketChannel sc : chnls) { sc.close(); }
 }
 Collection<SocketChannel> createChannels() {
  List<SocketChannel> list = new LinkedList<SocketChannel>();
  list.add(createChannel("http://tinyurl.com/23qct8", 80));
  list.add(createChannel("http://tinyurl.com/23qct8", 80));
  //...
  return list;
 }
 SocketChannel createChannel(String hostName, int port) {
  SocketChannel sc = SocketChannel.open();
  sc.configureBlocking(false);
  return sc;
 }
 void receive(SocketChannel x) {
  File f = new File("ReceivedData");
  FileOutputStream fos = new FileOutputStream(f,true);
  ByteBuffer dst = ByteBuffer.allocateDirect(1024);
  int numBytesRead = 0;
  while (numBytesRead >= 0) {
   numBytesRead = x.read(dst);
   fos.write(dst.array());
  }
  fos.close();
 }
 void send(SocketChannel x) {
  for (?) {
   ByteBuffer buf = ByteBuffer.allocateDirect(1024);
   buf.put((byte) 0xFF);
   buf.flip();
   int numBytesWritten = x.write(buf);
  }
 }
}
```

Figure 1: A simple program using APIs of interest.

The experimental results indicate that in order to produce reasonable specifications, the static analysis must employ sufficiently precise abstractions of aliases and event sequences. Based on experience with the prototype implementation, we discuss strengths and weaknesses of static analysis for specification mining. We conclude that this approach shows promise as a path to more effective specification mining tools.

2. OVERVIEW

Fig. 1 shows a simple Java program that uses `SocketChannel` objects. Our goal is to infer the pattern of `SocketChannel` API calls the program invokes on any individual object. Fig. 2 shows a partial specification of the `SocketChannel` API,[1] representing a desirable analysis output. The specification indicates that the program must *connect* a `SocketChannel` before using it. Connecting a channel entails a sequence of three operations: (1) configuring the channel's blocking mode, (2) requesting a connection, and (3) finishing the connection process by waiting for the connection to be established. Once the channel is connected, the program can invoke `read` and `write` in any order, and eventually, `close`.

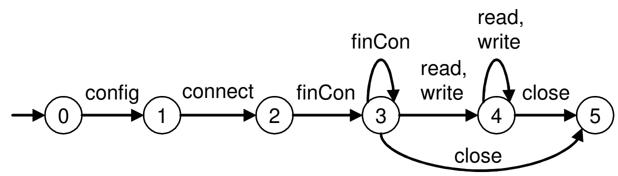


Figure 2: Partial specification for `SocketChannel`.

To extract this pattern from the example code, an analysis must deal with complex heap-allocated data structures in order to track the state of individual objects. Note that the method `createChannels` returns a collection containing an arbitrary number of dynamically allocated `SocketChannel` objects, which flow across procedure boundaries to other API calls. In order to make sense of the temporal sequence of operations on any individual channel, the analysis must employ precise alias analysis to track the sequence of operations on individual objects.

In addition to challenges with alias analysis, the specification inference must deal with a second difficult abstraction issue: tracking state related to a potentially *unbounded* sequence of events for each object. For example, the `receive` method of Fig. 1 invokes `x.read(dst)` in a `while` loop with unknown bounds.

2.1 Our Approach

Our approach consists of two phases: an *abstract-trace collection* phase, which accumulates abstractions of event sequences for abstract objects, using *abstract histories*, and a *summarization* phase, which consolidates the abstract histories and reduces noise.

2.1.1 Abstract-Trace Collection

We statically collect data regarding the event sequences for objects of a particular type. We use abstract interpretation [6], where an abstract value combines pointer information with an *abstract history*, a bounded representation of the sequence of events for a tracked object in the form of an automaton.

Our trace collection analysis is governed by two general parameters: (i) the heap abstraction and (ii) the history abstraction. Table 1 shows the abstract histories generated for the example program, varying the choice of heap abstraction and history abstraction.

The table columns represent two heap abstractions presented previously [9]; the *Base* abstraction, which relies on a flow-insensitive Andersen's pointer analysis [3], and the *APFocus* abstraction, which employs fairly precise flow-sensitive access-paths analysis. The table rows represent variations on the history abstraction. The history abstraction relies on an *extend* operator and a *merge* operator. In the table, we fix the extend operator to one that distinguishes past behavior (the *Past* relation of Section 4.2.3), and vary the choice of merge operator.

The merge operator controls the *join* used to combine histories that arise at the same program point but in different execution paths. The *Total* operator joins *all* histories that occur in a particular abstract program state. The *Exterior* operator joins only histories that share a common recent past, as will be described formally later.

In the table, specifications become less permissive (and more precise) as one moves right and/or down. That is, the combination Base/Past/Total is the most permissive, and APFocus/Past/Exterior is the least permissive. The results show that the analysis requires *both a rather precise aliasing and a rather precise merge operator* in order to approach the desired result.

[1] Figures in the paper use abbreviated method names.

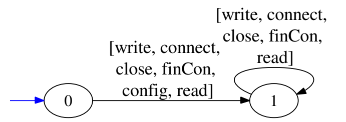
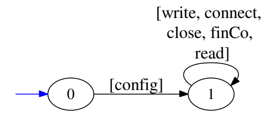
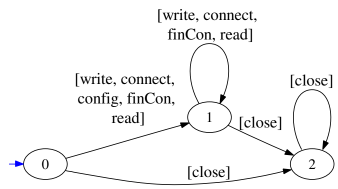
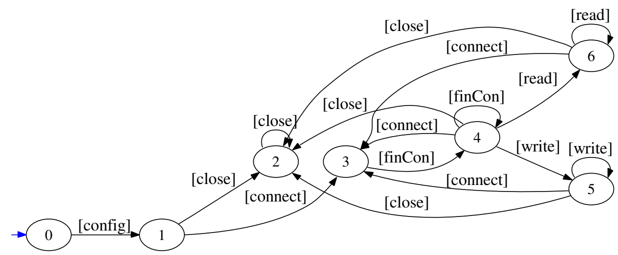
<table>
<tr><th></th><th>Base</th><th>APFocus</th></tr>
<tr><td>Past/
Total</td><td>
</td><td>
</td></tr>
<tr><td>Past/
Exterior</td><td>
</td><td>
</td></tr>
</table>

Table 1: Results of mining the running example with varying heap abstractions and merge algorithms.

2.1.2 Summarization

Abstract trace collection generates a set of abstract histories that overapproximates possible client behavior. However, some generated histories will admit spurious behavior (noise), either due to analysis imprecision or bugs in the client corpus.

The summarization phase employs statistical approaches to consolidate the collected abstract histories. In contrast to most previous work, which summarizes either raw event traces [5, 2] or event pairs [22, 20], our "raw data" (automata) already exhibit some structure resembling a candidate specification.

Our summarization phase exploits this structure to perform effective noise elimination and consolidation. In particular, we show a clustering algorithm to partition the abstract histories into groups that represent related scenarios. The approach eliminates noise from each cluster independently, allowing it to distinguish noise from interference between independent use cases.

Returning to the running example, we note that the least permissive abstract history (APFocus/Past/Exterior) contains a few edges that look spurious, such as repeated calls to `close` (state 2 self-loop) and repeated calls to `connect` (state 6 to state 3). In fact, these transitions will occur in the example program if the same `SocketChannel` appears twice in the collection; however, most likely the programmer does not intend for this to happen, and perhaps some invariant rules out this pathological case. When this abstract history is summarized with others that do not exhibit this particular pathology, the summarization algorithm will rule out the spurious edges, resulting in the specification of Fig. 2.

We further note that the quality of the input abstract histories limits the quality of the summarization output. It is hard to imagine any summarization algorithm producing the desired specification based on overly permissive input, such as the abstract history from Base/Past/Total.

3. PRELIMINARIES

In this section, we provide some basic definitions that we will use in the rest of the paper.

DEFINITION 3.1. *Given a finite set* Σ *of* input symbols*, a* finite automaton *over* alphabet Σ *is a tuple* $\mathcal{A} = (\Sigma, \mathcal{Q}, init, \delta, \mathcal{F})$*, where* $\mathcal{Q}$ *is a finite set of* states*,* $init \in \mathcal{Q}$ *is the* initial state*,* $\delta \colon \mathcal{Q}\times\Sigma \to 2^{\mathcal{Q}}$ *is the* transition function *and* $\mathcal{F} \subseteq \mathcal{Q}$ *is the set of* accepting states.

An automaton $\mathcal{A}$ is *deterministic* if for every $q \in \mathcal{Q}$ and $\sigma \in \Sigma$, $|\delta(q, \sigma)| \leq 1$. δ is extended to finite words in the usual way. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all words $\alpha \in \Sigma^*$ such that $\delta(init, \alpha) \cap \mathcal{F} \neq \emptyset$.

For an automaton state $q \in \mathcal{Q}$, we define $in_k(q) = \{\alpha \in \Sigma^k \mid \exists q' \in \mathcal{Q} : q \in \delta(q', \alpha)\}$. Similarly, $out_k(q) = \{\alpha \in \Sigma^k \mid \exists q' \in \mathcal{Q} : q' \in \delta(q, \alpha)\}$. In particular, $in_0 = out_0 = \{\epsilon\}$, where ϵ denotes the empty sequence. To ensure that for every $q \in \mathcal{Q}$ and every $k \geq 1$, $in_k(q), out_k(q) \neq \emptyset$, we extend Σ by some $\perp \notin \Sigma$ and view each state that has no predecessor (resp. successor) as having an infinite ingoing (resp. outgoing) sequence $\perp^\omega$.

DEFINITION 3.2 (QUOTIENT). *Let* $\mathcal{A} = (\Sigma, \mathcal{Q}, init, \delta, \mathcal{F})$ *be an automaton, and* $\mathcal{R} \subseteq \mathcal{Q} \times \mathcal{Q}$ *an equivalence relation on* $\mathcal{Q}$*, where* $[q]$ *denotes the equivalence class of* $q \in \mathcal{Q}$*. Then the* quotient automaton *is* $Quo_{\mathcal{R}}(\mathcal{A}) = (\Sigma, \{[q] \mid q \in \mathcal{Q}\}, [init], \delta', \{[q] \mid q \in \mathcal{F}\})$*, where* $\delta'([q], \sigma) = \{[q'] \mid \exists q'' \in [q] : q' \in \delta(q'', \sigma)\}$.

The quotient automaton is an automaton whose states consist of the equivalence classes of states of the original automaton. The outgoing transitions are then defined as the union of the outgoing transitions of all the states in the equivalence class (this might result in nondeterministic automata even if $\mathcal{A}$ is deterministic). It is easy to show that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(Quo_{\mathcal{R}}(\mathcal{A}))$.

In the following, the alphabet Σ consists of method calls (observable events) over the objects of the tracked type.

4. ABSTRACT TRACE COLLECTION

Our trace collection analysis produces "abstract histories", which summarize the event sequences of many possible concrete executions. The analysis-propagates a sound approximation of program state that tracks alias information and histories for each abstract object.

In the following, we describe the analysis in terms of a sound abstraction of an instrumented concrete semantics.

4.1 Concrete Instrumented Semantics

We define an instrumented concrete semantics that tracks the concrete trace of events for each concrete object. We refer to the concrete trace of events as the *concrete history* of the concrete object. We start with a standard concrete semantics for an imperative object-oriented language, defining a program state and evaluation of an expression in a program state.

Restricting our attention to reference types, the semantic domains are defined in a standard way as follows:

$$
\begin{array}{lcl}
L^\natural & \in & 2^{objects^\natural} \\
v^\natural & \in & Val = objects^\natural \cup \{null\} \\
\rho^\natural & \in & Env = VarId \to Val \\
\pi^\natural & \in & Heap = objects^\natural \times FieldId \to Val \\
state^\natural = \langle L^\natural, \rho^\natural, \pi^\natural \rangle & \in & States = 2^{objects^\natural} \times Env \times Heap
\end{array}
$$

where *objects*$^\natural$ is an unbounded set of dynamically allocated objects, *VarId* is a set of local variable identifiers, and *FieldId* is a set of field identifiers.

A *program state* keeps track of the set $L^\natural$ of allocated objects, an *environment* $\rho^\natural$ mapping local variables to values, and a mapping $\pi^\natural$ from fields of allocated objects to values.

In our instrumented semantics, each concrete object is mapped to a "concrete history" that records the sequence of events that has occurred for that object. Technically, we define the notion of a *history* which captures a regular language of event sequences.

DEFINITION 4.1. *A* history h *is a finite automaton* $(\Sigma, \mathcal{Q}, init, \delta, \mathcal{F})$*, where* $\mathcal{F} \neq \emptyset$*.* $\mathcal{F}$ *is also called the set of* current states. *We define the* traces represented by h, $Tr(h)$*, to be the language* $\mathcal{L}(h)$*.*

A *concrete history* $h^\natural$ is a special case of a history that encodes a single finite trace of events, that is, where $Tr(h^\natural)$ consists of a single finite trace of events. In Sec. 4.2 we will use the general notion of a history to describe a regular language of event sequences. We refer to a history that possibly describes more than a single trace of events as an *abstract history*.

EXAMPLE 4.2. *Fig. 3 shows examples of concrete histories occurring for a* `SocketChannel` *object of the example program at various points of the program. Fig. 4 and Fig. 5 show examples of abstract histories describing regular languages of events. In all figures, current states are depicted as double circles. Note that the automaton corresponding to an abstract history may be non-deterministic (e.g., as shown in Fig. 5).*

We denote the set of all concrete histories by $\mathcal{H}^\natural$. We augment every concrete state $\langle L^\natural, \rho^\natural, \pi^\natural \rangle$ with an additional mapping $his^\natural : L^\natural \rightharpoonup \mathcal{H}^\natural$ that maps an allocated object of the tracked type to its concrete history. A state of the instrumented concrete semantics is therefore a tuple $\langle L^\natural, \rho^\natural, \pi^\natural, his^\natural \rangle$.

Given a state $\langle L^\natural, \rho^\natural, \pi^\natural, his^\natural \rangle$, the semantics generates a new state $\langle L^{\natural\prime}, \rho^{\natural\prime}, \pi^{\natural\prime}, his^{\natural\prime} \rangle$ when evaluating each statement. We assume a standard interpretation for program statements updating $L^\natural$, $\rho^\natural$, and $\pi^\natural$. The $his^\natural$ component changes when encountering object allocations and observable events:

- *Object Allocation:* For a statement `x = new T()` allocating an object of the tracked type, a new (fresh) object $l_{new} \in objects^\natural \setminus L^\natural$ is allocated, and $his^{\natural\prime}(l_{new}) = h_0^\natural$, where $h_0^\natural = (\Sigma, \{init\}, init, \delta_0, \{init\})$ and δ_0 is a transition function that maps every state and event to an empty set. That is, the newly allocated object is mapped into the empty-sequence history.
- *Observable Events:* For a statement `x.m()` where $\rho^\natural(x)$ is of the tracked type T, the object $\rho^\natural(x)$ is mapped to a new concrete history $extend^\natural(h^\natural, m)$, where $h^\natural = his^\natural(\rho^\natural(x))$ and $extend^\natural : \mathcal{H}^\natural \times \Sigma \to \mathcal{H}^\natural$ is the concrete *extend transformer* that adds exactly one new state to $h^\natural$, in the natural way, to reflect the call to $m()$.

Statement	Concrete History
sc = open()	$\to\odot$
sc.config	$\to\bigcirc\xrightarrow{cfg}\odot$
sc.connect	$\to\bigcirc\xrightarrow{cfg}\bigcirc\xrightarrow{cnc}\odot$
sc.finCon	$\to\bigcirc\xrightarrow{cfg}\bigcirc\xrightarrow{cnc}\bigcirc\xrightarrow{fin}\odot$
…	
sc.finCon	$\to\bigcirc\xrightarrow{cfg}\bigcirc\xrightarrow{cnc}\bigcirc\xrightarrow{fin}\ldots\xrightarrow{fin}\odot$
x.read	$\to\bigcirc\xrightarrow{cfg}\bigcirc\xrightarrow{cnc}\bigcirc\xrightarrow{fin}\ldots\xrightarrow{fin}\bigcirc\xrightarrow{rd}\odot$
…	
x.read	$\to\bigcirc\xrightarrow{cfg}\bigcirc\xrightarrow{cnc}\bigcirc\xrightarrow{fin}\ldots\xrightarrow{fin}\bigcirc\xrightarrow{rd}\ldots\xrightarrow{rd}\odot$
sc.close	$\to\bigcirc\xrightarrow{cfg}\bigcirc\xrightarrow{cnc}\bigcirc\xrightarrow{fin}\ldots\xrightarrow{fin}\bigcirc\xrightarrow{rd}\ldots\xrightarrow{rd}\bigcirc\xrightarrow{cl}\odot$

Figure 3: Example of concrete histories for an object of type SocketChannel in the example program.

Fig. 3 shows the evolution of concrete histories for an object in the example program. Each concrete history records the sequence of observable events (method calls) upon the `SocketChannel` during a particular execution. Note that the length of a concrete history is a priori unknown, as events may occur in loops.

4.2 Abstract Semantics

The instrumented concrete semantics uses an unbounded description of the program state, resulting from a potentially unbounded number of objects, each with a potentially unbounded history. In this section we describe an abstract semantics that conservatively represents the instrumented semantics with various degrees of precision and cost.

4.2.1 Abstract States

Following [9], we base the abstraction on a global *heap graph*, obtained through a flow-insensitive, partially context-sensitive subset-based may points-to analysis [3]. This provides a partition of the *objects*$^\natural$ set into abstract objects, each partition with a unique name called an *instance key*.

The heap graph representation of our motivating example contains a single instance key for type `SocketChannel`, representing all the objects allocated in `createChannel`.

An abstract program state consists of a set of tuples, called "factoids." A *factoid* is a tuple $\langle o, \textit{heap-data}, h \rangle$, where

- o is an instance key.
- *heap-data* consists of multiple components describing heap properties of o (as in [9]).
- h is the abstract history representing the traces observed for o until the corresponding execution point.

An abstract state can contain multiple factoids for the same instance key o, representing different alias contexts and abstract histories.

The *heap-data* component of the factoid is crucial for precision; we adopt the *heap-data* abstractions of [9]. Intuitively, the heap abstraction relies on the combination of a preliminary scalable (e.g. flow-insensitive) pointer analysis and selective predicates indicating access-path aliasing, and information on object uniqueness. Informally, a factoid with instance key o, and with *heap-data* $= \{unique = true, must = \{x.f\}, mustNot = \{y.g\}, may = true\}$ represents a program state in which there exists exactly one object named o, such that $x.f$ must evaluate to point to o, $y.g$ must *not* evaluate to point to o, and there *may* be other pointers to o not represented by these access-paths . Crucially, the tracking of *must point-to* information allows *strong updates* [4] when propagating dataflow information through a statement.

Due to space constraints, we must elide further exposition of the heap abstraction and refer the reader to [9].

While a concrete history describes a unique trace, an abstract history typically encodes multiple traces as the language of the automaton. Different abstractions consider different history automata (e.g. deterministic vs. non-deterministic) and different restrictions on the current states (e.g. exactly one current state vs. multiple current states). We denote the set of abstract histories by $\mathcal{H}$. The remainder of this section considers semantics and variations of history abstractions.

4.2.2 Abstract Semantics

An abstract semantics for the history is defined via the following:

- An abstract extend transformer, $extend : \mathcal{H} \times \Sigma \to \mathcal{H}$, and
- A merge operator $\bigsqcup : 2^{\mathcal{H}} \to 2^{\mathcal{H}}$ which generates a new set of abstract histories that overapproximates the input set.

In the abstract semantics, the abstract history component for a fresh object is initialized to $h_0^\natural$ (the empty-sequence history). When an observable event occurs, the semantics updates the relevant histories using the $extend$ operator.

As long as the domain of abstract histories is bounded, the abstract analysis is guaranteed to terminate. However, in practice, it can easily suffer from an exponential blowup due to branching control flow. The merge operator will mitigate this blowup, accelerating convergence. Specifically, at control flow join points, all factoids that represent the same instance key and have identical heap-data are merged. Such factoids differ only in their abstract histories, i.e., they represent different execution paths of the same abstract object in the same aliasing context.

Soundness. We design the abstraction to keep track of (at least) all the traces, or concrete histories, produced by the code base. We denote the set of all concrete histories possibly generated by a code base C by $\mathcal{H}_C^\natural$. Similarly, we denote the set of all abstract histories generated by the analysis of C by $\mathcal{H}_C$. The analysis is *sound* if for every concrete history $h^\natural$ in $\mathcal{H}_C^\natural$ there exists some abstract history in $\mathcal{H}_C$ whose set of traces includes the single concrete trace represented by $h^\natural$. I.e.,

$$\bigcup_{h^\natural \in \mathcal{H}_C^\natural} Tr(h^\natural) \subseteq \bigcup_{h \in \mathcal{H}_C} Tr(h).$$

Soundness is achieved by making sure that every reachable (instrumented) concrete state *istate*$^\natural$ is represented by some reachable abstract state *istate*, meaning that for every object $o^\natural \in L^\natural$ there exists a factoid $\langle o, \textit{heap-data}, h\rangle$ in *istate* that provides a sound representation of $o^\natural$. This is a factoid whose *heap-data* component fulfills the conditions described in [9], and in addition h is a sound representation of $his^\natural(o^\natural)$, i.e. $Tr(his^\natural(o^\natural)) \subseteq Tr(h)$. Soundness of the extend transformer and of the merge operator ensure that the analysis is sound.

DEFINITION 4.3. *An abstract extend transformer* $extend$ *is* sound*, if whenever* $Tr(h^\natural) \subseteq Tr(h)$ *then for every* $\sigma \in \Sigma$, $Tr(extend^\natural(h^\natural, \sigma)) \subseteq Tr(extend(h, \sigma))$.

DEFINITION 4.4. *A merge operator* $\bigsqcup$ *is* sound*, if for every set of abstract histories* $H \subseteq \mathcal{H}$, $\bigcup_{h\in H} Tr(h) \subseteq \bigcup_{h \in \bigsqcup H} Tr(h)$.

Precision. The analysis is *precise* if it does not introduce additional behaviors that do not appear in the code-base, i.e.

$$\bigcup_{h^\natural \in \mathcal{H}_C^\natural} Tr(h^\natural) = \bigcup_{h \in \mathcal{H}_C} Tr(h).$$

Remark. In practice, instead of considering the traces represented by *all* the abstract histories generated by the analysis, we consider the *prefix-closures* of the history automata at the exit-points of the program, obtained by setting $\mathcal{F}' = \mathcal{Q}$ (i.e., all the states are considered accepting). Since the history automata associated with the exit-points are "maximal", then the set of observed traces is maintained when we restrict ourselves to their prefix-closures.

4.2.3 History Abstractions

We present a parameterized framework for history abstractions, based on intuition regarding the structure of API specifications.

Quotient-Based Abstractions. In practice, automata that characterize API specifications are often simple, and further admit simple characterizations of their states (e.g. their ingoing or outgoing sequences). Exploiting this intuition, we introduce abstractions based on quotient structures of the history automata, which provide a general, simple, and in many cases precise, framework to reason about abstract histories.

Given an equivalence relation $\mathcal{R}$, and some *merge criterion*, we define the quotient-based abstraction of $\mathcal{R}$ as follows.

- The *extend transformer* appends the new event σ to the current states, and constructs the quotient of the result. More formally, let $h = (\Sigma, \mathcal{Q}, \textit{init}, \delta, \mathcal{F})$. For every $q_i \in \mathcal{F}$ we introduce a fresh state, $n_i \notin \mathcal{Q}$. Then $extend(h, \sigma) = Quo_{\mathcal{R}}(h')$, where $h' = (\Sigma, \mathcal{Q} \cup \{n_i \mid q_i \in \mathcal{F}\}, \textit{init}, \delta', \{n_i \mid q_i \in \mathcal{F}\})$ with $\delta'(q_i, \sigma) = \delta(q_i, \sigma) \cup \{n_i\}$ for every $q_i \in \mathcal{F}$, and $\delta'(q', \sigma') = \delta(q', \sigma')$ for every $q' \in \mathcal{Q}$ and $\sigma' \in \Sigma$ such that $q' \notin \mathcal{F}$ or $\sigma' \neq \sigma$.
- The *merge operator* first partitions the set of histories based on the given *merge criterion*. Next, the merge operator constructs the union of the automata in each partition, and returns the quotient of the result.

It can be shown that for every equivalence relation $\mathcal{R}$ and merge criterion, the quotient-based abstraction w.r.t. $\mathcal{R}$ is sound.

To instantiate a quotient-based abstraction, we next consider options for the requisite equivalence relation and merge criteria.

Past-Future Abstractions. In many cases, API usages have the property that certain sequences of events are always preceded or followed by the same behaviors. For example, a `connect` event of `SocketChannel` is always followed by a `finishConnect` event. This means that the states of the corresponding automata are characterized by their ingoing and/or outgoing behaviors. As such, we consider quotient abstractions w.r.t. the following parametric equivalence relation.

DEFINITION 4.5 (PAST-FUTURE RELATION). *Let* q_1, q_2 *be history states, and* $k_1, k_2 \in \mathbb{N}$. *We write* $(q_1, q_2) \in R[k_1, k_2]$ *iff* $in_{k_1}(q_1); out_{k_2}(q_1) \cap in_{k_1}(q_2); out_{k_2}(q_2) \neq \emptyset$, *i.e.* q_1 *and* q_2 *share both an ingoing sequence of length* k_1 *and an outgoing sequence of length* k_2.

For example, consider the abstract history depicted in Fig. 6(a). States 2 and 4 (marked by arrows) are equivalent w.r.t. $R[1, 0]$ since $in_1(2) = in_1(4) = \{\texttt{cnc}\}$ and $out_0(2) = out_0(4) = \{\epsilon\}$.

We will hereafter focus attention on the two extreme cases of the past-future abstraction, where either k_1 or k_2 is zero. Recall that

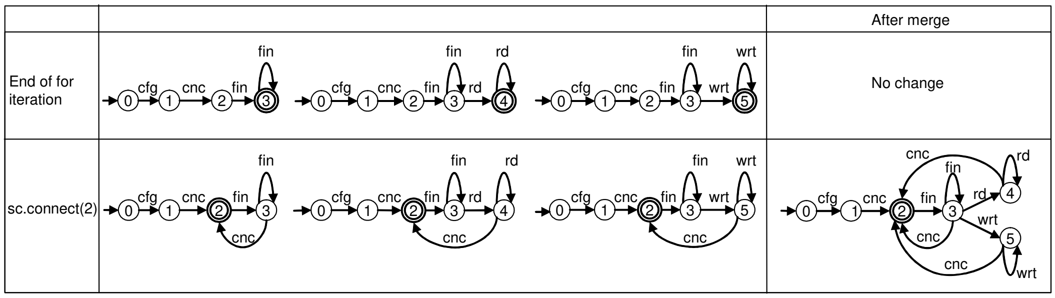


Figure 4: Abstract interpretation with past abstraction (Exterior merge).

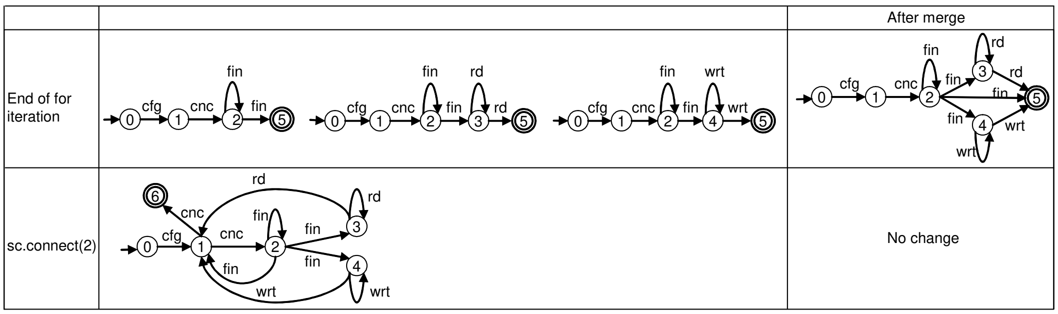


Figure 5: Abstract interpretation with future abstraction (Exterior merge).

$in_0(q) = out_0(q) = \{\epsilon\}$ for every state q. As a result, $R[k, 0]$ collapses to a relation that considers ingoing sequences of length k. We refer to it as $\mathcal{R}^k_{past}$, and to the abstraction as the *k-past abstraction*. Similarly, $R[0, k]$ refers to outgoing sequences of length k, in which case we also refer to it as $\mathcal{R}^k_{future}$. We refer to the corresponding abstraction as the *k-future abstraction*. Intuitively, analysis using the k-past abstraction will distinguish patterns based only on their recent past behavior, and the k-future abstraction will distinguish patterns based only on their near future behavior. These abstractions will be effective if the recent past (near future) suffices to identify a particular behavioral sequence.

Merge Criteria. Having defined equivalence relations, we now consider merge criteria to define quotient-based abstractions. A merge criterion will determine when the analysis should collapse abstract program states, thus potentially losing precision, but accelerating convergence.

We consider the following merge schemes.

- *Total:* all histories are merged into one.
- *Exterior:* the histories are partitioned into subsets in which all the histories have compatible initial states and compatible current states. Namely, histories h_1 and h_2 are merged only if (a) $(init_1, init_2) \in \mathcal{R}$; and (b) for every $q_1 \in \mathcal{F}_1$ there exists $q_2 \in \mathcal{F}_2$ s.t. $(q_1, q_2) \in \mathcal{R}$, and vice versa.

Intuitively, the total criterion forces the analysis to track exactly one abstract history for each "context" (i.e. alias context, instance key, and program point).

The exterior criterion provides a less aggressive alternative, based on the intuition that the distinguishing features of a history can be encapsulated by the features of its initial and current states. The thinking follows that if histories states differ only on the characterization of intermediate states, merging them may be an attractive option to accelerate convergence without undue precision loss.

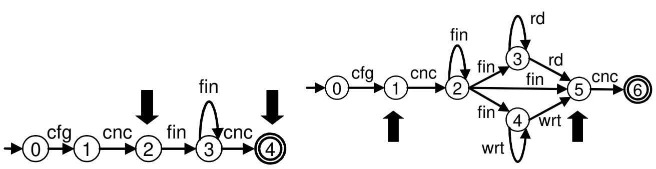
Example. Fig. 4 presents abstract histories produced during the analysis of the single instance key in our running example, using the 1-past abstraction with exterior merge. The first row describes the histories observed at the end of the first iteration of the `for` loop of `example()`. These all hold abstract histories for the same instance key at the same abstract state. Each history tracks a possible execution path of the abstract object.

Although these histories refer to the same instance key and alias context, exterior merge does not apply since their current states are not equivalent. The second row shows the result of applying the extend transformer on each history after observing a `connect` event. The intermediate step, before the quotient construction, for the automaton on the left is depicted in Fig. 6(a). There, and in all other cases as well, the new state is equivalent to an existing state according to the 1-past relation; a state with `connect` as its incoming event already exists in each automaton. As a result, extend simply adds the new transitions and adds no new states.

After observing this event, the resulting three histories meet the exterior merge criterion, and are therefore combined. The analysis discards the original histories and proceeds with the merged one which overapproximates them.

Fig. 5 presents the corresponding abstract histories using the 1-future abstraction with exterior merge (in fact, in this case total merge behaves identically). Unlike the case under the past abstraction, merge applies at the end of the first loop iteration, since the initial and current states are equivalent under the 1-future relation. As a result, the analysis continues with the single merged history. The second row shows the result of applying the extend transformer on it after observing a `connect` event.

Fig. 6(b) presents the intermediate step in which the merged abstract history is extended by `connect`, before the quotient is constructed. An outgoing transition labelled `connect` is added from state 5 (the previous current state) to a new state, making state 5 share a future with state 1. Thus states 1 and 5 are merged.



(a) (b)

Figure 6: (a) Past abstraction step; and (b) Future abstraction step, before quotient construction.

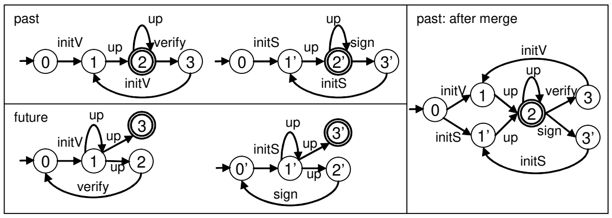


Figure 7: Past abstraction vs. future abstraction.

Nondeterminism. It is easy to verify that the quotient structure of a deterministic automaton w.r.t. $\mathcal{R}^k_{past}$ is deterministic. This ensures that the k-past abstraction always produces deterministic automata, as demonstrated by Fig. 4. On the other hand, when the future parameter is nontrivial (i.e. $k_2 \neq 0$), nondeterminism can result during the quotient construction. For example, note that in Figure 5, all the automata are non-deterministic.

Precision. If automata satisfy the following structural property, then we can prove that the past-future abstraction is fully precise:

DEFINITION 4.6. *An automaton $\mathcal{A}$ has the (k_1, k_2)-past-future property if for every $q_1 \neq q_2 \in \mathcal{Q}$, $in_{k_1}(q_1); out_{k_2}(q_1) \cap in_{k_1}(q_2); out_{k_2}(q_2) = \emptyset$. This implies that every sequence of length $k_1 + k_2$ is linked to a unique automaton state.*

PROPOSITION 4.7 (PRECISION GUARANTEE). *If $\bigcup_{h^\natural \in \mathcal{H}^\natural_C} Tr(h^\natural)$ is accepted by some automaton that has the (k_1, k_2)-past-future property, then the (k_1, k_2)-past-future abstraction with exterior merge is precise.*

When the precision precondition is not met, different choices of k_1, k_2 in the past-future abstraction can lead to different results:

EXAMPLE 4.8. *The first row of Fig. 7 presents two histories produced while using the 1-past abstraction to track an abstract object that uses the `Signature` API. The history on the left uses the `verify` feature of the API, while the history on the right uses `sign`. The current states of these two histories, states 2 and $2'$, are compatible ($in_1(2) = in_1(2') = \{\texttt{update}\}$), and the histories are therefore merged into the history presented in Fig. 7 on the right. In particular, states 2 and $2'$ are merged. As a result, the relation between an invocation of `initVerify` (resp. `initSign`) and a later invocation of `verify` (resp. `sign`) is lost.*

When using the 1-future abstraction, on the other hand, the corresponding abstract histories, depicted in the second row of Fig. 7, are not compatible since their initial states are not compatible ($out_1(0) = \{\texttt{initVerify}\}$, while $out_1(0') = \{\texttt{initSign}\}$), and are therefore not merged, preventing the precision loss.

Of course, increasing the parameters k_1 and k_2 makes the abstraction more precise, but may negatively impact convergence.

5. SUMMARIZATION

The abstract trace collection produces automata that overapproximate the actual behavior. However, the trace collection output may represent spurious behavior due to at least three sources of noise:

- **Analysis Imprecision:** The output of the abstract interpretation is an over-approximation that may include behavior from infeasible paths.
- **Bugs in Training Corpus:** Programs in the training corpus may contain a (hopefully small) number of incorrect usages.
- **Unrestricted Methods:** Some API methods (e.g. side-effect free methods) may not play a role in the intended API specification, but may still appear in the collected abstract traces.

To deal with unrestricted methods, one could leverage *component-side* techniques to analyze the API implementation, identify side-effect-free methods, and exclude them from consideration [18]. Similarly, we could apply component-side analysis to exclude spurious patterns which lead to violations of simple safety or liveness properties. We elide further discussion of such techniques due to space constraints.

To deal with the other sources of noise, we turn to statistical techniques inspired by approaches such as z-ranking [7] and the ranking of [20]. Statistical techniques distinguish signal from noise according to sample frequency. A crucial factor concerns what relative weight to assign to each sample.

We observe that each static occurrence of a usage pattern represents some thought and work by a programmer, while each dynamic occurrence typically represents an iteration of a loop counter. We assign weights to patterns based on a conjecture that the *number of times an API usage pattern appears in the code* provides a more meaningful metric than the number of times that code executes [2].

Most previous work on statistical trace analyses considered raw traces consisting of raw event streams [5, 2] or event pairs [22, 20]. In contrast, our work summarizes samples that already represent summarized abstract traces, represented as automata. In this section, we present new approaches to statistical summarization that exploit the structure already present in these automata.

Due to its statistical nature, summarization does not maintain the soundness guarantee of the trace collection phase. Namely, it might erroneously identify correct behaviors as spurious ones, and remove them.

In the sequel, we assume without loss of generality, that the observed traces are given via a set $\mathcal{I}$ of deterministic finite automata (if nondeterministic automata were produced, we add a determinization step). The output of summarization consists of a ranked set of $k \leq |\mathcal{I}|$ automata, where each of the k represents a candidate API specification.

5.1 Union Methods

Naive Union. The naive approach outputs the union of all the automata in $\mathcal{I}$ as the API specification, without any noise reduction. This approach treats all traces uniformly, regardless of their frequency. Moreover, it does not distinguish between different ways in which the API is used.

Weighted Union. A better straightforward statistical approach uses a weighted union of the input automata to identify and eliminate infrequent behaviors. Specifically, we form the union automaton for all input automata, labelling each transition with the count of the number of input automata which contain it. Given

[2] An empirical evaluation of this conjecture falls outside the scope of this paper, but would be an interesting direction for future work.

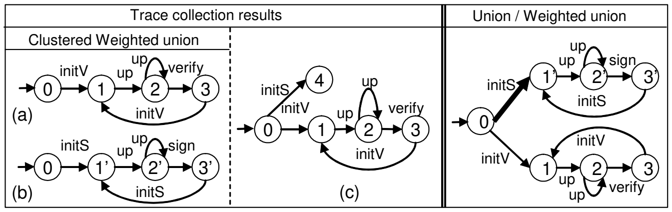


Figure 8: Summarization: Clustered Weighted Union Vs. Weighted Union.

this labelled automaton, one can apply any number of heuristics to discard transitions with low weights. Our implementation takes a threshold parameter f ($0 \leq f \leq 1$) and discards any transitions whose weight is less than f times the number of input automata.

5.2 Clustering

When a code-base contains several independent patterns of API usage, these patterns may interfere to defeat union-based noise reduction. Instead, we partition the input into clusters of "similar" automata, and eliminate noise in each cluster independently.

We use a simple clustering algorithm based on a notion of automata *inclusion*. Automaton A *includes* automaton B iff $\mathcal{L}(A) \supseteq \mathcal{L}(B)$. The *include* relation induces a partial order on the set of automata. Each "maximal" element (automaton) w.r.t. this order represents a cluster consisting of all the automata included in it. Our algorithm forms clusters based on inclusion, and then applies the weighted union technique independently in each cluster.

EXAMPLE 5.1. *Consider Fig. 8. Each of the automata (a) and (b) represents a possible usage of the* `Signature` *API in some code-base. Assume that each of them was observed numerous times. A weighted union of them with any reasonable threshold will return the right most automaton in Fig. 8, where the two usage patterns are combined. A clustered union, on the other hand, will identify that these are two different usage patterns, and will return (a) and (b) as two clusters.*

Assume further, that the code-base also produced the automaton (c) of Fig. 8. Automaton (c) refers to the same usage pattern as (a), but contains an additional transition. This transition is erroneous in this particular usage pattern, although it is not erroneous in the global view of the API. In the weighted union, this simply increases the weight of the bold edge by 1*, but it is not recognized as an error. Our clustered weighted union, on the other hand, recognizes that (c) belongs to the cluster of (a), and as a result it identifies and removes the erroneous transition.*

Note that after transitions are removed, the *include* relation can change as disparate clusters sometimes converge. As such, we iterate the entire process, starting from the clustering, until reaching a fixpoint. As a post-pass, an entire cluster can be removed as noise based on further statistical analysis.

6. EXPERIMENTAL RESULTS

We have implemented a prototype of our analysis based on the WALA analysis framework [19] and the typestate analysis framework of [9]. Our analysis builds on a general Reps-Horwitz-Sagiv (RHS) IFDS tabulation solver implementation [17]. We extended the RHS solver to support dynamic changes and merges in the set of dataflow facts. The pointer analysis adds one-level of call-string context to calls to various library factory methods, `arraycopy`, and `clone` statements, which tend to badly pollute pointer flow precision if handled without context-sensitivity. The system uses a substantial library of models of native code behavior for the standard libraries.

Num	Benchmark	Classes	Methods	Bytecodes	Contexts	Clients
1	aamfetch	635	2544	246284	3316	2
2	bobalice	259	1318	71048	1917	2
3	crypty	450	2138	127130	2794	1
4	flickrapi	123	423	26607	527	2
5	ganymed	121	649	49232	919	4
6	j2ns	944	4817	399402	6570	5
7	javacup	373	2000	122592	2981	2
8	javasign	111	473	45670	740	11
9	jbidwatcher	64	525	18717	269	9
10	jfreechart	654	2644	250718	3457	18
11	jlex	89	317	25261	382	2
12	jpat-p	374	2043	141649	5570	1
13	JPDStore	109	359	23040	418	2
14	js-chap13	661	2795	259273	3770	6
15	privatray	175	665	56543	876	1
16	tinysql	701	3019	277881	3980	2
17	tvla	643	2572	249243	3355	3

Table 2: Benchmarks.

6.1 Benchmarks

Table 2 lists the benchmarks used in this study. Each of the benchmarks `bobalice`, `js-chap13`, and `j2ns`, is a set of examples taken from a book on Java security [16]. `flickrapi` is an open source program providing a wrapper over flickr APIs, as well as some utilities using it. `ganymed` is a library implementing the SSH-2 protocol in pure Java; the library comes with examples and utility programs that use it. `javacup` and `jlex` are a parser generator and lexical analyzer, respectively, for Java. `jbidwatcher` is an online auction tool. `jfreechart` is a Java chart library. `tinysql` is a lightweight Java SQL engine. `tvla` is a static analysis framework.

The table reports size characteristics restricted to methods discovered by on-the-fly call-graph construction. The call graph includes methods from both the application and the libraries; for many programs the size of the program analyzed is dominated by the standard libraries. The table also reports the number of (method) contexts in the call graph (the context-sensitivity policy models some methods with multiple contexts). The last column shows the number of client programs for each benchmark.

Using these clients, we applied our prototype to mine specifications for a number of APIs, as described in more detail in the next section. Our implementation employs standard automata minimization, and our results always refer to minimized automata.

Our most precise solvers (APFocus/Past/Exterior and APFocus/-Future/Exterior) run in less than 30 minutes per benchmark. Our less precise solvers run in about half the time. This performance seems reasonable for non-interactive mining of a code base.

6.2 Results

Our evaluation focuses first on three dimensions for abstract trace collection:

- The heap abstraction (Sec. 4.2.1): Base vs. APFocus
- The history abstraction (Sec. 4.2.3): Past vs. Future
- The merge criteria (Sec. 4.2.3): Total vs. Exterior merge

Table 3 characterizes the specifications generated by our analysis, varying the abstractions along these three dimensions. Each row summarizes the result for a specific API across a number of benchmarks.

Some APIs appear in several separate benchmarks, while others appear in several programs contained within the same bench-

	Base/Past/Total			Base/Past/Ext			Base/Future/Ext			APF/Past/Total			APF/Past/Ext			APF/Future/Ext		
API	**states**	**edges**	**avg. degree**	**states**	**edges**	**avg. degree**	**states**	**edges**	**avg. degree**	**states**	**edges**	**avg. degree**	**states**	**edges**	**avg. degree**	**states**	**edges**	**avg. degree**
Auth	2	3	1.50	2	3	1.5	2	3	1.5	2	2	1.00	2	2	1.00	2	2	1.00
Channel	2	6	3.00	3	6	2.00	3	6	2.00	3	3	1.00	3	3	1.00	3	3	1.00
ChannelMgr	2	11	5.50	5	18	3.60	6	19	3.17	4	7	1.75	5	9	1.80	5	9	1.80
Cipher	1	5	5.00	4	14	3.50	6	12	2.00	7	10	1.43	7	10	1.43	7	10	1.43
Connection	3	12	4.00	4	12	3.00	4	12	3.00	5	7	1.40	5	7	1.40	5	7	1.40
KeyAgreement	2	5	2.50	4	6	1.50	4	6	1.5	4	3	0.75	4	3	0.75	4	3	0.75
LineAndShape	3	12	4.00	6	15	2.50	6	15	2.50	6	8	1.33	6	8	1.33	6	8	1.33
MsgDigest	1	2	2.00	2	2	1.00	2	2	1.00	2	2	1.00	2	2	1.00	2	2	1.00
Photo	1	12	12.00	1	12	12.00	1	8	8.00	8	8	1.00	8	8	1.00	8	8	1.00
PrintWriter	1	3	3.00	2	3	1.50	2	3	1.50	6	11	1.83	3	5	1.67	3	5	1.67
Session	2	7	3.50	5	10	2.00	5	10	2.00	5	4	0.80	5	4	0.80	5	4	0.80
Signature	2	8	4.00	5	12	2.40	5	12	2.40	4	6	1.50	4	6	1.50	4	6	1.50
TransportMgr	9	24	2.67	2	19	9.50	8	27	3.38	9	26	2.89	9	24	2.67	9	24	2.67
URLConnection	2	9	4.50	4	10	2.5	3	6	2	4	7	1.75	NA	NA		NA	NA	
Average			**4.08**			**3.46**			**2.57**			**1.39**			**1.33**			**1.33**
Std dev			**2.54**			**3.22**			**1.71**			**0.56**			**0.52**			**0.52**

Table 3: Characteristics of our mined specifications with varying data collectors. For every mined specification DFA, we show the number of states, edges, and the density of the DFA.

mark. The `Auth` and `Photo` APIs are used in benchmark 4. `Channel`, `ChannelManager`, `Connection`, `Session`, and `TransportManager` are used in benchmark 5. `Cipher` is used in benchmarks 1, 3, 14, and 15. `KeyAgreement` is used in benchmark 2. `LineAndShapeRenderer` is used in benchmark 10. `MessageDigest` is used in benchmarks 1, 13, and 15. `PrintWriter` is used in benchmarks 7 and 11. `Signature` is used in 6 and 8. `URLConnection` is used in benchmark 9.

Each column of the table corresponds to a combination of a heap abstraction, history abstraction, and merge criterion. When using Total merge, we only show results for Past history abstraction; results for Future would be similar under this aggressive merge criterion. All results in the table reflect the Naive Union summarization (Sec. 5.1), which preserves all information collected by the trace collectors. This allows to compare the quality of different abstract trace collectors without interference from summarization effects.

The table reports, for each mined specification, the number of states and transitions, and the average degree (number of outgoing edges) of states in the specification. Intuitively, the degree of a node represents the number of possible legal operations from a given state. Since all specifications in the table overapproximate client behavior, a smaller degree represents a better specification since it admits fewer spurious behaviors. Note that the different overapproximations may be incomparable in terms of the languages they accept; that is, we cannot, in general, rank the mined specifications based on a simulation ordering. Note also that average degree is a relative metric; its absolute value depends on the number of observable events in the specification.

The results show across the board that precise alias analysis is significant; the mined specifications appear significantly more permissive under Base aliasing than under APFocus. Exterior merge improves over total merge frequently when using Base aliasing, and occasionally under APFocus aliasing. When using the most precise APFocus aliasing and exterior merge, the distinction between past and future abstractions vanishes in these experiments, although they behave significantly differently under Base aliasing.

For some specifications, we were able to track the usage pattern manually by inspecting the client code. For others, the complexity of the client code (or even lack of Java source code) prevented us from understanding the client API usage based on inspection.

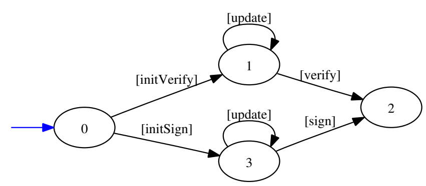


Figure 9: Mined Specification for java.security.Signature, obtained with APFocus/Past/Exterior.

Based on manual inspection, we were able to verify that for 5 out of the 14 APIs, the most precise analysis generated the ideal specification, even with the naive union summarization. These APIs appear in bold in the table.

We additionally collected specifications with weighted summarization for each benchmark. We do not report densities obtained by weighted summarization, as the specification density will depend directly on the threshold parameter provided as input. We note based on inspection that a user-provided threshold of 1/2 for the weighted union algorithm yields improved specifications for several APIs. In practice, we expect a user would provide feedback to iteratively tune the threshold as desired.

We also ran the cluster-based summarization for all specifications. For several APIs, clustering correctly identified a number of independent usage patterns of the API in our code base. In particular, the specification obtained for `Cipher` using the naive union collector was polluted by calls to irrelevant methods. Using the combination of clustering and weighted algorithms, we obtained the "ideal" specification expected by a human.

Fig. 9 shows the output of our tool for the `Signature` API. This API was mined using APFocus/Past/Exterior collector, and summarized using the naive union summarization. Note that the specification correctly disambiguates two use cases, *verify* and *sign*. An approach based on event pairs (e.g. [20, 22]) could not distinguish these two cases.

Fig. 10 shows the output of our tool for the `ganymed Session` API under two collector settings, and summarized using the naive union summarization. This figure shows a typical qualitative difference between Base aliasing and APFocus.

A small gallery of mined specifications appears in an informal online supplement to this paper [10].

6.3 Discussion

Our experiments indicate that having both a precise-enough heap abstraction and a precise-enough history abstraction are required to be able to mine a reasonable specification.

Without such abstractions, the collected abstract histories might deteriorate to a point in which no summarization algorithm will recover the lost information. For example, the specification mined for the `Photo` API using the Base heap abstraction has a single state. This means that the specification does not contain any temporal information on the ordering of events. (Similarly for `PrintWriter` under Base/Past/Total.)

6.3.1 Soundness

All of the results in Table 3 were obtained when our analysis was run to completion, and are therefore guaranteed to be an over-approximation of the behavior present in the analyzed code base. In contrast, it is also possible to employ our analysis with a predetermined timeout (or with e.g., a small limited heap size). In such cases, the specification obtained using the analysis will not over-approximate code base behavior, but may still help understand some behaviors. For example, when running on TVLA, we mined a partial but interesting description of the way `tvla.Engine` is used in the code base.

6.3.2 Limitations

Our prototype shows encouraging results, but due to several limitations, does not yet suffice for deployment in an industrial tool.

Our implementation currently considers all methods of an API as equally interesting. In general, this pollutes specifications with calls to pure methods that do not change the state of the component. When library code is available, one might analyze the library to identify pure methods and treat them specially in both abstract trace collection and summarization. In the absence of library code, we envision a feedback loop involving user input, specifying methods that should be ignored.

In some cases, the specifications mined by our approach are too detailed, and track distinctions that hold no interest to the end user. For example, the specification we mine for `PrintWriter` records some artificial temporal ordering between `print` and `println`. We'd expect these problems to resolve themselves with a larger input corpus; if not, a practical tool would probably resort to user feedback to refine results.

Our current implementation does not scale to code bases larger than roughly a few tens of thousands of lines in reasonable time and space (depending on properties of the dataflow solution). The scaling problem is fundamental to all whole program analyses, and does not stem primarily from the particular history abstractions introduced in this paper. In the future, we plan to explore how this technique could be turned into a modular one in the spirit of [23], which we believe is a crucial step for a practical implementation.

Our current prototype restricts itself to specifications involving a single object; however, many interesting specifications involve multiple types and objects. The ideas presented in this paper can apply to components that involve multiple objects, but the scalability and precision questions remain open.

Despite these limitations, we are encouraged by the results obtained with our current implementation, which show the strength of our heap and history abstractions, as well as our summarization algorithms. We also expect these abstractions to be useful in the context of other analyses that track temporal sequences.

7. RELATED WORK

Dynamic Analysis. When it is feasible to run a program with adequate coverage, dynamic analysis represents the most attractive option for specification mining, since dynamic analysis does not suffer from the difficulties inherent to abstraction.

Cook and Wolf [5] consider the general problem of extracting an FSM model from an event trace, and reduce the problem to the well-known *grammar inference* [11] problem. Cook and Wolf discuss algorithmic, statistical, and hybrid approaches, and present an excellent overview of the approaches and fundamental challenges. This work considers mining automata from uninterpreted event traces, attaching no semantic meaning to events.

Ammons et al. [2] infer temporal and data dependence specifications based on dynamic trace data. This work applies sophisticated probabalistic learning techniques to boil traces down to collections of finite automata which characterize the behavior.

Whaley et al. [21] present a two-phased approach to mining temporal API interfaces, combining a static component-side safety check with a dynamic client-side sequence mining. This work presents several insights on how to refine results, based on side-effect free methods, and partitioning methods based on how they access fields. We plan to incorporate these insights into a future version of our analysis.

The Perracotta tool [22] addressed challenges in scaling dynamic mining of temporal properties to large code bases. This tool mines traces for two-event patterns with an efficient algorithm, and relies on heuristics to help identify interesting patterns.

Livshits and Zimmerman [13] mine a software repository revision history to find small sets of methods whose usage may be correlated. This analysis is simple and scalable; in contrast to ours, it does not consider temporal ordering nor aliasing. In a second (dynamic) phase, the system checks whether candidate temporal patterns actually appear in representative program traces. Our analysis technology could perhaps be employed in a similar architecture to provide a more effective first phase of mining.

A number of projects mine specifications in the form of *dynamic invariant detection*. Daikon [8] instruments a running program and infers invariants based on values of variables in representative program traces, typically discovering method preconditions, postconditions, and loop invariants. DIDUCE [12] combines invariant detection and checking in a single tool, aimed to help diagnose failures. As a program runs, DIDUCE maintains a set of hypothesized invariants, and reports violations of these invariants as they occur.

Component-side Static Analysis. In component-side static analysis, a tool analyzes a component's implementation, and infers a specification that ensures the component does not fail in some predetermined way, such as by raising an exception. In contrast, client-side mining produces a specification that represents the usage scenarios in a given code-base. The two approaches are complementary, as demonstrated in [21].

Recent years have seen a number of sophisticated component-side static analyses. Alur et al. [1] use Angluin's algorithm together with a model-checking procedure to learn a permissive interface of a given component. Gowri et al. [15] use static mining to learn a permissive first-order specification involving objects, their relationships, and their internal states.

Client-side Static Analysis. A few papers have applied static analysis to client-side specification mining.

Engler et al. [7] use various static analyses to identify common

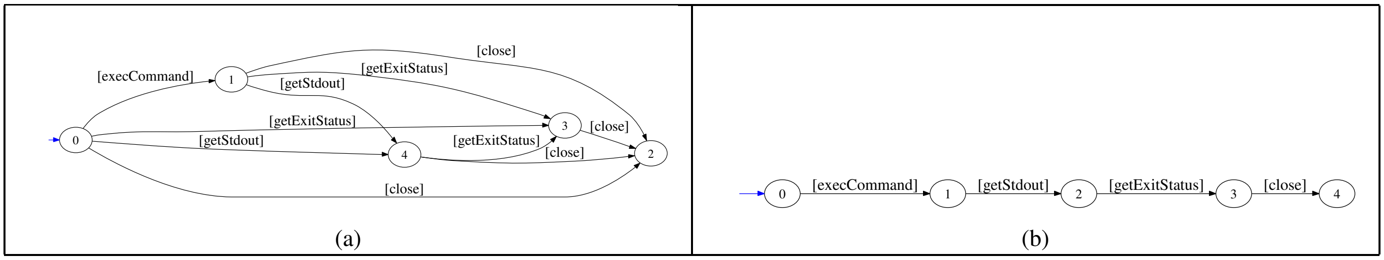


Figure 10: Session API with Past/Exterior merge and (a) Base aliasing, (b) APFocus aliasing.

program behaviors, and consider deviations from these behaviors as bugs. Their approach automatically establishes certain invariants as likely *beliefs*, and the tool searches for bugs by finding code sequences that violate these invariants. The tool searches for invariants based on a set of standard templates, and filters potential specifications with a statistical measure (z-ranking). Their approach has been highly effective in finding bugs in system code.

Weimer and Necula [20] use a simple, lightweight static analysis to infer simple specifications from a given codebase. Their insight is to use exceptional program paths as negative examples for correct API usage. We believe that our approach could also benefit from using exceptional paths as negative examples. Weimer and Necula learn specifications that consist of pairs of events $\langle a, b\rangle$, where a and b are method calls, and do not consider larger automata. They rely on type-based alias analysis, and so their techniques should be much less precise than ours. On the other hand, their paper demonstrates that even simple techniques can be surprisingly effective in finding bugs.

Mandelin et al. [14] use static analysis to infer a sequence of code (*jungloid*) that shows the programmer how to obtain a desired target type from a given source type (a *jungloid query*). This code-sequence is only checked for type-safety and does not address the finer notion of typestate.

8. CONCLUSION

To our knowledge, this paper presents the first study attempting client-side temporal API mining with static analysis beyond trivial alias analysis and history abstractions. Static analysis improves coverage over dynamic analysis both by exploring all paths for a single program, and by expanding the corpus of code amenable to automated analysis. We plan to conduct further research into modular analysis techniques and improved summarization heuristics, to move closer to practical application of this technology.

9. REFERENCES

[1] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. *SIGPLAN Not.*, 40(1):98–109, 2005.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, New York, NY, USA, 2002. ACM Press.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, May 1994. (DIKU report 94/19).

[4] D. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 296–310, New York, NY, 1990. ACM Press.

[5] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, 1998.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[7] 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 *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, New York, NY, USA, 2001. ACM Press.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[9] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, New York, NY, USA, 2006. ACM Press.

[10] Gallery of mined specification. http://tinyurl.com/23qct8 or http://docs.google.com/View?docid=ddhtqgv6_10hbczjd.

[11] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[12] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. May 2002.

[13] V. B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 13th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-13)*, pages 296–305, Sept. 2005.

[14] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, New York, NY, USA, 2005. ACM Press.

[15] M. G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 77–96, New York, NY, USA, 2005. ACM Press.

[16] M. Pistoia, D. Reller, D. Gupta, M. Nagnur, and A. K. Ramani. *Java 2 Network Security*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, August 1999.

[17] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 49–61, 1995.

[18] A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *VMCAI'05: Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2005.

[19] WALA: The T. J. Watson Libraries for Analysis. http://wala.sourceforge.net.

[20] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *TACAS*, 2005.

[21] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 218–228. ACM Press, July 2002.

[22] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM Press.

[23] G. Yorsh, E. Yahav, and S. Chandra. Symbolic summarization with applications to typestate verification. Technical report, Tel Aviv University, 2007. www.cs.tau.ac.il/∼gretay.