The SAFE Experience

Eran Yahav and Stephen Fink

Abstract We present an overview of the techniques developed under the SAFE project. The goal of SAFE was to create a practical lightweight framework to verify simple properties of realistic Java applications. The work on SAFE covered a lot of ground, starting from typestate verification techniques [18, 19], through inference of typestate specifications [34, 35], checking for absence of null derefences [26], automatic resource disposal [13], and an attempt at modular typestate analysis [42]. In many ways, SAFE represents a modern incarnation of early ideas on the use of static analysis for software reliability (e.g., [21]). SAFE went a long way in making these ideas applicable to real properties of real software, but applying them at the scale of modern framework-intensive software remains a challenge. We are encouraged by our experience with SAFE, and believe that the technique developed in SAFE can serve as a solid basis for future work on practical verification technology.

1 Introduction

Statically checking if a program satisfies specified safety properties (e.g., [21, 14, 29, 28, 8, 12, 5, 22, 20, 4, 30, 11, 17]) can help identify defects early in the development cycle, thus increasing productivity, reducing development costs, and improving quality and reliability.

Typestate [37] is an elegant framework for specifying a class of temporal safety properties. Typestates can encode correct usage rules for many common libraries and application programming interfaces (APIs) (e.g. [39, 1]). For example, typestate can express the property that a Java program should not read data from `java.net.Socket` until the socket is connected.

The SAFE project focused on various aspects of typestate verification and inference with the ultimate goal of being able to verify nontrivial properties over realistic programs.

Eran Yahav
Technion, Israel, e-mail: yahave@cs.technion.ac.il

Stephen Fink
IBM T.J. Watson Research Center e-mail: sjfink@us.ibm.com

1.1 Challenges

We focus on sound analysis; if the verifier reports no violation, then the program is guaranteed to satisfy the desired properties. However, if the verifier reports a potential violation, it might not correspond to an actual program error. Imprecise analysis can lead a verifier to produce "false alarms": reported problems that do not indicate an actual error. Users will quickly reject a verifier that produces too many false positives.

Scaling to Real Software. While sophisticated and precise analyses can reduce false positives, such analyses typically do not scale to real programs. Real programs often rely on large and complex supporting libraries, which the analyzer must process in order to reason about program behavior.

Aliasing. There is a wide variety of efficient flow-insensitive may-alias (pointer) analysis techniques (e.g. [10, 24, 36]) that scale to fairly large programs. These analyses produce a statically bounded (abstract) representation of the program's runtime heap and indicate which abstract objects each pointer-valued expression in the program may denote. Unfortunately, these scalable analyses have a serious disadvantage when used for verification. They require the verifiers to model any operation performed through a pointer dereference conservatively as an operation that may or may not be performed on the *possible* target abstract objects identified by the pointer analysis – this is popularly known as a "weak update" as opposed to a "strong update" [6].

To support strong updates and more precise alias analysis, we present a framework to check typestate properties by solving a flow-sensitive, context-sensitive dataflow problem on a *combined domain of typestate and pointer information*. As is well-known [9], a combined domain may yield improved precision compared to separate analyses. Furthermore, the combined domain allows the framework to *concentrate computational effort on alias analysis only where it matters to the typestate property*. This concentration allows more precise alias analysis than would be practical if applied to the whole program.

Using a domain that combines precise aliasing information with other property-related information is a common theme in our work ([17, 33, 18, 19, 34, 35, 26, 26, 42]). Our precise treatment of aliasing draws some ideas from our experience with shape analysis [32]. In the rest of this paper we describe the combined domains we used for typestate verification (Sec. 2), inference of typestate specifications (Sec. 3), and verifying the absence of null derefences (Sec. 4).

Getting Specifications. Typestate verification requires typestate specifications. It is not always easy, or possible, to obtain such formal specifications from programmers. This raises the research question of whether such specification can be obtained automatically. Indeed, much research has addressed *mining specifications* directly from code [1, 2, 7, 38, 40, 41, 25, 16, 23, 27, 15].

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 approach we have taken in SAFE is to obtain specifications using *static client-side specification mining*. The idea in *client-side* specification mining is to examine the ways client programs use that API. An effective static client-side specification mining shares many of the challenges with typestate verification. In particular, it also requires relatively precise treatment of aliasing to track the sequence of events that may occur at runtime for every object of interest.

2 Typestate Verification

Consider the simple program of Fig. 1. We would like to verify that this program uses Sockets in a way that is consistent with the typestate property of Fig. 2.

As mentioned earlier, we would like our verifier to be sound in the presence dynamic allocation and aliasing. We now informally present a number of abstractions in attempt to verify our example program.

```
1  open(Socket s) { s.connect();}
2  talk(Socket s) { s.getOutputStream()).write(''hello''); }
3  dispose(Socket s) { s.close(); }
4  main() {
5    Socket s = new Socket(); //S          { <S,init> }
6    open(s);                              { <S,init>, <S,conn> }
7    talk(s);                              { <S,init>, <S,conn>, <S,err> }
8    dispose(s);
9  }
```

Fig. 1 A simple program using `Socket`.

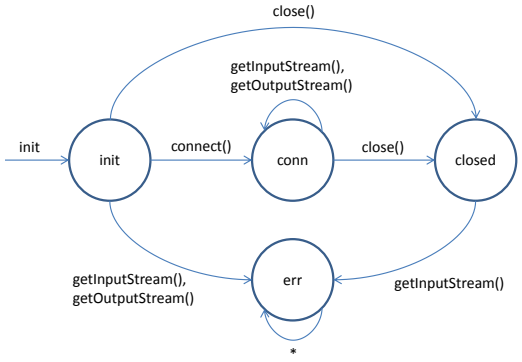


Fig. 2 Partial specification for a Socket.

Naming Objects in an Unbounded Heap. The idea behind all of our abstraction is to combine aliasing information with typestate information. The first component of our abstraction is a global *heap graph*, obtained through a flow-insensitive, context-sensitive subset based may points-to analysis [3]. This is fairly standard and provides a partition of the set *objects*[♮] into abstract objects. In this discussion, we define an *instance key* to be an abstract object name assigned by the flow-insensitive pointer analysis. The heap graph provides for an access path e, the set of instance keys it *may* point-to and also the set of access paths that may be aliased with e.

Base Abstraction. Our initial abstraction attempt is an abstraction referred to as a *base abstraction* in [18]. This abstraction tracks the typestate state for every allocation site in the program. The results for this abstraction are shown as annotations in Fig. 1. We use a tuple $\langle Obj, St\rangle$ to denote the fact that the abstract state contains an object Obj that is in the state St. The abstract state is a set of tuples of this form.

In the example of Fig. 1, a Socket object is allocated in line 5, leading to an abstract state $\{\langle S, init\rangle\}$. After the call to `open(s)`, the abstract state contains two possibilities $\{\langle S, init\rangle, \langle S, connected\rangle\}$. This is because when performing the call `s.connect()`, we do not have sufficient information to perform a *strong update*. The abstract object S potentially represent multiple objects of type socket. Invoking `s.connect()` does not make *all* of these objects change their state to connected. Therefore, to be sound, we have to assume that it is possible for the state to contain other objects, that may be pointed to by s, and that remain in their initial state *init*. This sound assumption makes our analysis report a false alarm when invoking `talk(s)`.

Unique Abstraction. The example of Fig. 1 actually allocates a single Socket object, and there is no justification for the loss of precision in such simple (and common) cases. To deal with cases in which an allocation site is only used to allocate a single object, we introduce a slightly refined abstraction, tracking tuples of the form $\langle Obj, Un, St\rangle$ representing the fact that the abstract state contains an object Obj in the state St, and a boolean flag Un denotes whether this is the only object allocated at the site Obj.

```
1  open(Socket s) { s.connect();}
2  talk(Socket s) { s.getOutputStream()).write(''hello''); }
3  dispose(Socket s) { s.close(); }
4  main() {
5    Socket s = new Socket(); //S          { <S,U,init> }
6    open(s);                              { <S,U,conn> }
7    talk(s);                              { <S,U,conn> }
8    dispose(s);
9  }
```

Fig. 3 A simple program using `Socket` with Unique abstraction.

The result of using this abstraction for the simple example program are shown in Fig. 3. Tracking the fact that the allocation site S only allocates a unique object allows us to perform a strong update on $\langle S, unique, init\rangle$ when `connect(s)` is in-

voked. This results in the single tuple $\langle S, unique, conn \rangle$, and thus with successful verification of this example. However, it is easy to see that whenever an allocation site is executed inside a loop, the unique abstraction may be insufficient and result in false alarms similar to the ones we had with the base abstraction. We therefore introduce a family of abstractions that integrate typestate information and flow-sensitive must (and may) points-to information enabling to handle destructive updates on non-unique abstract objects.

2.1 Integrating Typestate and Must Points-to Information

The simple abstractions presented earlier are used as preceding phases of the SAFE analyzer to dismiss simple cases. The real strength of SAFE comes from the integrated verifier that tracks must points-to information using access paths and combines it with typestate information.

Technically, our abstract semantics uses a combination of two representations to abstract heap information: (i) a global heap-graph representation encoding the results of a flow insensitive points-to analysis; (ii) enhanced flow-sensitive must points-to information integrated with typestate checking.

2.1.1 Parameterized Typestate Abstraction

Our parameterized abstract representation uses tuples of the form: $\langle o, unique, typestate, AP_{must}, May, AP_{mustNot} \rangle$ where:

- o is an instance key.
- *unique* indicates whether the corresponding allocation site has a single concrete live object.
- *typestate* is the typestate of instance key o.
- AP_{must} is a set of access paths that must point-to o.
- *May* is true indicates that there are access paths (not in the must set) that may point to o.
- $AP_{mustNot}$ is a set of access paths that do not point-to o.

This parameterized abstract representation has four dimensions, for the *length* and *width* of each access path set (must and must-not). The length of an access path set indicates the maximal length of an access path in the set, similar to the parameter k in k-limited alias analysis. The width of an access path set limits the number of access paths in this set.

An abstract state is a set of tuples. We observe that a conservative representation of the concrete program state must obey the following properties:

(a) An instance key can be indicated as unique if it represents a single object for this program state.

(b) The access path sets (the must and the must-not) do not need to be complete. This does not compromise the soundness of the staged analysis due to the indication of the existence of other possible aliases.
(c) The must and must-not access path sets can be regarded as another heap partitioning which partitions an instance key into the two sets of access paths: those that a) must alias this abstract object, and b) definitely do not alias this abstract object. If the must-alias set is non-empty, the must-alias partition represents a single concrete object.
(d) If $May = false$, the must access path is complete; it contains all access paths to this object.

```
1  class SocketHolder {  Socket s;  }
2  Socket makeSocket() {
3    return new Socket(); // A
4  }
5  open(Socket t) {    { <A,¬U,init,{},May,{}> }
6    t.connect();      { <A,¬U,init,{},May,{ ¬t }>, <A,¬U,conn,{ t },May,{}> }
7  }
8  talk(Socket s) {
9    s.getOutputStream()).write(hello);
10            { <A,¬U,init,{},May,{ ¬g,¬s }>, <A,¬U,conn,{ g,s },May,{}>}
11 }
12 main() {
13   Set<SocketHolder> set = new HashSet<SocketHolder>();
14   while( ) {
15      SocketHolder h = new SocketHolder();
16      h.s = makeSocket();
17      set.add(h);                          { <A,U,init,{ h.s },May,{}> }
18   }
19   for (Iterator<SocketHolder> it = set.iterator();  ) {
20      Socket g = it.next().s;
21      open(g);
22      talk(g);  { <A,¬U,init,{},May,{ ¬g }>, <A,¬U,conn,{ g },May,{}>}
23    }
24 }
```

Fig. 4 A program using `Socket`s stored inside a collection.

The* focus *operation. A key element of SAFE's abstraction is the use of a *focus* operation [32], which is used to dynamically (during analysis) make distinctions between objects that the underlying basic points-to analysis does not distinguish.

We now describe the focus operation, which improves the precision of the analysis. As a motivating example, consider the statement `t.connect()` in line 6 of the example. Since this statement is invoked on sockets coming from the iterator loop of lines 20-23, we have an incoming tuple representing all of the sockets in the collection, and, hence, we cannot apply a strong update to the tuple, which can subsequently cause a false positive.

The *focus* operation replaces the single tuple with two tuples, one representing the object that `t` points to, and another tuple to represent the remaining sockets. Formally, consider an incoming tuple $\langle o, unique, typestate, AP_{must}, true, AP_{mustNot} \rangle$ at an observable operation $e.op()$, where $e \notin AP_{must}$, but e may point to o (accord-

ing to the flow-insensitive points-to solution). The analysis replaces this tuple by the following two tuples:

$$\langle o, unique, typestate, AP_{must} \cup \{e\}, true, AP_{mustNot} \rangle$$
$$\langle o, unique, typestate, AP_{must}, true, AP_{mustNot} \cup \{e\} \rangle$$

In our example under consideration, the statement `t.connect()` is reached by the tuple $\langle A, \neg U, init, \emptyset, May, \emptyset \rangle$. Focusing replaces this tuple by the following two tuples:

$$\langle A, \neg U, init, \{t\}, true, \emptyset \rangle$$
$$\langle A, \neg U, init, \emptyset, true, \{t\} \rangle$$

The invocation of `t.connect()` is analyzed after the focusing. This allows for a strong update on the first tuple and no update on the second tuple resulting in the two tuples:

$$\langle A, \neg U, conn, \{t\}, true, \emptyset \rangle$$
$$\langle A, \neg U, init, \emptyset, true, \{t\} \rangle$$

We remind the reader that the *unique* component tuple merely indicates if multiple objects allocated at the allocation site o may be simultaneously alive. A tuple such as $\langle A, false, conn, \{t\}, true, \emptyset \rangle$, however, represents a single object at this point, namely the object pointed to by t, which allows us to use a strong update.

The analysis applies this *focus* operation whenever it would otherwise perform a weak update for a typestate transition. Thus, focus splits the dataflow facts tracking the two typestates that normally result from a weak update.

3 Static 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 overestimates the intended use cases.

The SAFE approach provides static analysis for client-side mining, applied to API specifications for object-oriented libraries.

Consider the program of Fig. 5. We would like to extract the specification of how this program uses objects of type `SocketChannel`. 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:

```
1  class SocketChannelClient {
2   void example() {
3    Collection<SocketChannel> channels = createChannels();
4    for (SocketChannel sc : channels) {
5     sc.connect(new InetSocketAddress("tinyurl.com/23qct8",80));
6     while (!sc.finishConnect()) {
7      // ... wait for connection ...
8     }
9     if (?) {
10     receive(sc);
11    } else {
12     send(sc);
13    }
14   }
15   closeAll(channels);
16  }
17  void closeAll(Collection<SocketChannel> chnls) {
18    for (SocketChannel sc : chnls) { sc.close(); }
19  }
20  Collection<SocketChannel> createChannels() {
21   List<SocketChannel> list = new LinkedList<SocketChannel>();
22   list.add(createChannel("http://tinyurl.com/23qct8", 80));
23   list.add(createChannel("http://tinyurl.com/23qct8", 80));
24   //...
25   return list;
26  }
27  SocketChannel createChannel(String hostName, int port) {
28   SocketChannel sc = SocketChannel.open();
29   sc.configureBlocking(false);
30   return sc;
31  }
32  void receive(SocketChannel x) {
33   File f = new File("ReceivedData");
34   FileOutputStream fos = new FileOutputStream(f,true);
35   ByteBuffer dst = ByteBuffer.allocateDirect(1024);
36   int numBytesRead = 0;
37   while (numBytesRead >= 0) {
38    numBytesRead = x.read(dst);
39    fos.write(dst.array());
40   }
41   fos.close();
42  }
43  void send(SocketChannel x) {
44   for (?) {
45    ByteBuffer buf = ByteBuffer.allocateDirect(1024);
46    buf.put((byte) 0xFF);
47    buf.flip();
48    int numBytesWritten = x.write(buf);
49   }
50  }
51 }
```

Fig. 5 A simple program using APIs of interest.

- **Aliasing.** Objects from the target API may flow through complex heap-allocated data structures. For example, objects in the program of Fig. 5 are passed through Java collections.
- **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. For example, Fig. 6 shows a sample concrete history for the program of Fig. 5.

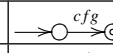
Statement	Concrete History
sc = open()	
sc.config	
sc.connect	
sc.finCon	
...	
sc.finCon	
x.read	
...	
x.read	
sc.close	

Fig. 6 Example of concrete histories for an object of type SocketChannel in the example program.

- **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.

In this paper we focus on the abstract trace collection phase, details about summarization can be found in [34, 35]. Experimental results in [34, 35], 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.

3.1 Abstract Trace Collection

The abstract trace collection requires abstraction on two unbounded dimensions: (i) abstraction of heap-allocated objects; (ii) abstraction of event sequences.

The abstraction we use for specification mining is based on the abstraction used for verification as described in Sec. 2. Both verifying a typestate property, and mining it, require precise reasoning on the sequences of events that can occur during program execution. The main difference is that mining typestate properties requires recording traces of events, where typestate verification only requires tracking of the state in a known typestate property.

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 (described below).
- 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 [18], as described in Sec. 2. Technically, the *heap-data* component of the factoid uses tuples of the form: $\langle \textit{unique}, AP_{\textit{must}}, \textit{May}, AP_{\textit{mustNot}} \rangle$ as described earlier. 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 $\textit{heap-data} = \{\textit{unique} = \textit{true}, \textit{must} = \{x.f\}, \textit{mustNot} = \{y.g\}, \textit{may} = \textit{true}\}$ 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* [6] when propagating dataflow information through a statement.

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 $\mathscr{H}$.

The remainder of this section considers semantics and variations of history abstractions.

Abstracting Histories. 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 $\mathscr{R}$, and some *merge criterion*, we define the quotient-based abstraction of $\mathscr{R}$ as follows.

- The abstraction h_0 of the empty-sequence history is $\mathit{Quo}_{\mathscr{R}}(h_0^\natural) = h_0$, i.e. the empty-sequence history.
- The *extend transformer* appends the new event σ to the current states, and constructs the quotient of the result. More formally, let $h = (\Sigma, \mathscr{Q}, \textit{init}, \delta, \mathscr{F})$. For every $q_i \in \mathscr{F}$ we introduce a fresh state, $n_i \notin \mathscr{Q}$. Then $\textit{extend}(h, \sigma) = \mathit{Quo}_{\mathscr{R}}(h')$, where $h' = (\Sigma, \mathscr{Q} \cup \{n_i \mid q_i \in \mathscr{F}\}, \textit{init}, \delta', \{n_i \mid q_i \in \mathscr{F}\})$ with $\delta'(q_i, \sigma) = \delta(q_i, \sigma) \cup \{n_i\}$ for every $q_i \in \mathscr{F}$, and $\delta'(q', \sigma') = \delta(q', \sigma')$ for every $q' \in \mathscr{Q}$ and $\sigma' \in \Sigma$ such that $q' \notin \mathscr{F}$ 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.

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.

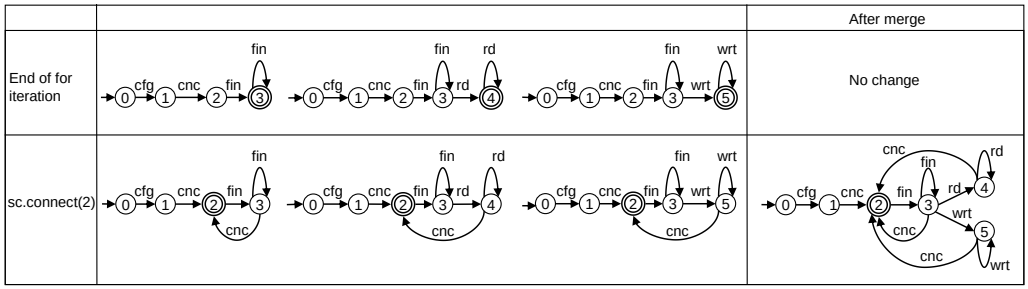


Fig. 7 Abstract interpretation with past abstraction (Exterior merge).

Definition 3.1 (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.*

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 $in_0(q) = out_0(q) = \{\varepsilon\}$ 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 $\mathscr{R}^k_{past}$, 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 $\mathscr{R}^k_{future}$. 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

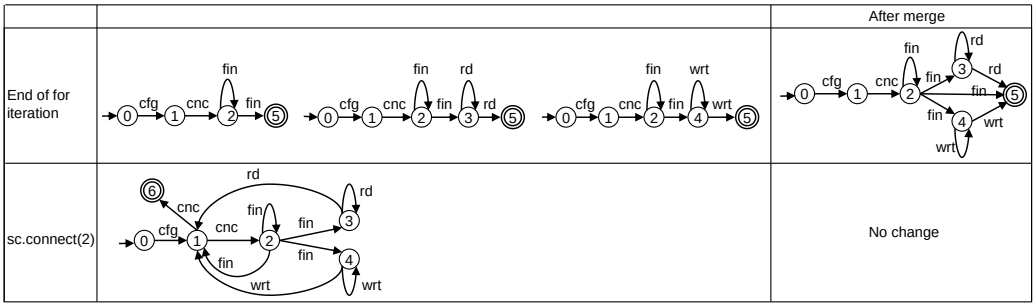


Fig. 8 Abstract interpretation with future abstraction (Exterior merge).

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.

- *None:* each history comprises a singleton set in the partition. This scheme is most precise, but is impractical, as it results in an exponential blowup.
- *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) $(init_1, init_2) \in \mathscr{R}$; and (b) for every $q_1 \in \mathscr{F}_1$ there exists $q_2 \in \mathscr{F}_2$ s.t. $(q_1, q_2) \in \mathscr{R}$, 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. 7 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.

Fig. 8 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.

4 Verifying Dereference Safety

Null-pointer dereferences represent a significant percentage of defects in Java applications. Furthermore, a null dereference is often a symptom of a higher-level problem; warning the programmer about a potential null dereference may help in exposing the more subtle problem. The general approach presented in the previous sections can be used to address the problem of verifying the safety of pointer dereferences in real Java programs.

We present a scalable analysis via lazy scope expansion. Unlike most existing bug-finding tools for detecting null dereferences our analysis is *sound*.

Our abstract domain is a product of three domains: (i) the abstract domain used for the may-alias analysis, (ii) the abstract domain used for the must-alias analysis, and (iii) a set AP_{nn} of access paths that are guaranteed to have a non-null value. We guarantee that the abstract domain is finite by placing a (parameterized) limit on the size of the AP_{nn} sets and on the maximal length of the access paths that they may contain.[1] We refer to the size of the access path set AP_{nn} as the *width* of AP_{nn} and to the maximal length of an access path $ap \in AP_{nn}$ as the *length* of AP_{nn}.

Note that domains (i) and (ii) above *are the same ones* used for typestate verification and specification mining in Sections 2 and 3.

For details on how elements of the third domain are updated, see [26]. Here, we wish to focus on another aspect, which is the notion of *expanding scopes*.

The idea is to break the verification problem into modular sub-problems. Specifically, our analysis operates on program *fragments*, and gradually expands the analysis scope in which a fragment is considered when additional context information is required. While this idea has been presented in [26] in the context of verifying dereference safety, it is applicable to all of the other analyses we presented here, and in particular to the typestate verification and typestate mining of the previous sections.

[1] The length of access path $\langle v, \langle f_1, \ldots, f_k \rangle \rangle$ is defined to be $k+1$.

Expanding-Scope Analysis. We present a *staged* analysis that adapts the *cost* of the analysis to the difficulty of the verification task. Our analysis breaks the verification problem into multiple subproblems and adapts the analysis of each subproblem along two dimensions: the *precision* dimension and the *analysis-scope* dimension. Our analysis adapts the *precision* (and thus the expected cost) of the abstract interpretation [9] to the difficulty of verifying the subproblem. In this aspect, it is similar to the staging in [18].

The novelty of our approach lies in its ability to adapt the *scope* of the analyzed program fragment to the difficulty of the verification task. Unlike existing staged techniques, which analyze whole programs (e.g., [18]), our analysis operates on program *fragments*. The basic idea, inspired by Rountev et. al. [31], is to break the program into fragments and analyze each fragment separately, making conservative assumptions about the parts of the program that lie outside the fragment. However, if the property cannot be verified under the conservative context assumptions, our approach provides for gradually *expanding the scope* of the analyzed fragment.

The premise of this approach is that a large percentage of the potential points of failure in a program can be verified by (i) using a scalable imprecise analysis that conservatively approximates context information, and (ii) employing more precise analyses that consider a limited scope, which may be expanded as needed.

Our approach is based on the principle of expanding scopes; it applies this principle to the problem of dereference safety, which is particularly challenging due to its dependence on aliasing information.

Another approach to modular analysis, that is based on symbolic representation of procedure summaries, can be found in [42].

5 Conclusion

In this paper, we surveyed some of the techniques developed in the SAFE project. Despite the sophisticated techniques used, the end result is that scalability of the approach remains limited. One of the main difficulties remains the sound treatment of large libraries. This is becoming more of a hurdle as software is becoming heavily based on framework code. We believe that practical success lies in the combination of static and dynamic techniques, as well as in moving to higher level languages that enable modular reasoning.

References

1. ALUR, R., CERNY, P., MADHUSUDAN, P., AND NAM, W. Synthesis of interface specifications for java classes. *SIGPLAN Not. 40*, 1 (2005), 98–109.
2. AMMONS, G., BODIK, R., AND LARUS, J. R. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2002), ACM Press, pp. 4–16.

3. ANDERSEN, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, May 1994. (DIKU report 94/19).
4. ASHCRAFT, K., AND ENGLER, D. Using programmer-written compiler extensions to catch security holes. In *Proc. IEEE Symp. on Security and Privacy* (Oakland, CA, May 2002).
5. BALL, T., AND RAJAMANI, S. K. Automatically validating temporal safety properties of interfaces. In *SPIN 2001: SPIN Workshop* (2001), LNCS 2057, pp. 103–122.
6. CHASE, D., WEGMAN, M., AND ZADECK, F. Analysis of pointers and structures. In *Proc. ACM Conf. on Programming Language Design and Implementation* (New York, NY, 1990), ACM Press, pp. 296–310.
7. COOK, J. E., AND WOLF, A. L. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol. 7*, 3 (1998), 215–249.
8. CORBETT, J., DWYER, M., HATCLIFF, J., PASAREANU, C., ROBBY, LAUBACH, S., AND ZHENG, H. Bandera: Extracting finite-state models from Java source code. In *Proc. Intl. Conf. on Software Eng.* (June 2000), pp. 439–448.
9. COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *Proc. ACM Symp. on Principles of Programming Languages* (New York, NY, 1979), ACM Press, pp. 269–282.
10. DAS, M. Unification-based pointer analysis with directional assignments. 35–46. In *Conference on Programming Language Design and Implementation (PLDI)*.
11. DAS, M., LERNER, S., AND SEIGLE, M. Esp: Path-sensitive program verification in polynomial time. In *PLDI* (2002), pp. 57–68.
12. DELINE, R., AND FÄHNDRICH, M. Enforcing high-level protocols in low-level software. In *Proc. ACM Conf. on Programming Language Design and Implementation* (June 2001), pp. 59–69.
13. DILLIG, I., DILLIG, T., YAHAV, E., AND CHANDRA, S. The CLOSER: Automating resource management in Java. In *ISMM '08: International Symposium on Memory Management* (2008). to appear.
14. DWYER, M. B., AND CLARKE, L. A. Data flow analysis for verifying properties of concurrent programs. In *Proc. Second ACM SIGSOFT Symp. on Foundations of Software Engineering* (New Orleans, LA, December 1994), pp. 62–75.
15. ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. 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* (New York, NY, USA, 2001), ACM Press, pp. 57–72.
16. ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering 27*, 2 (Feb. 2001), 99–123.
17. FIELD, J., GOYAL, D., RAMALINGAM, G., AND YAHAV, E. Typestate verification: Abstraction techniques and complexity results. In *SAS '03: 10th International Static Analysis Symposium* (2003), vol. 2694, Springer, pp. 439–462.
18. FINK, S., YAHAV, E., DOR, N., RAMALINGAM, G., AND GEAY, E. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis* (2006), ACM, pp. 133–144. * best paper award.
19. FINK, S. J., YAHAV, E., DOR, N., RAMALINGAM, G., AND GEAY, E. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology 17*, 2 (2008), 1–34.
20. FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for java. In *Proc. ACM Conf. on Programming Language Design and Implementation* (Berlin, June 2002), pp. 234–245.
21. FOSDICK, L. D., AND OSTERWEIL, L. J. Data flow analysis in software reliability. *ACM Comput. Surv. 8* (September 1976), 305–330.
22. FOSTER, J. S., TERAUCHI, T., AND AIKEN, A. Flow-sensitive type qualifiers. In *Proc. ACM Conf. on Programming Language Design and Implementation* (Berlin, June 2002), pp. 1–12.
23. HANGAL, S., AND LAM, M. S. Tracking down software bugs using automatic anomaly detection.

24. HEINTZE, N., AND TARDIEU, O. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. 254–263. In *Conference on Programming Language Design and Implementation (PLDI)*.
25. LIVSHITS, V. B., AND ZIMMERMANN, T. 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)* (Sept. 2005), pp. 296–305.
26. LOGINOV, A., YAHAV, E., CHANDRA, S., FINK, S., RINETZKY, N., AND NANDA, M. G. Verifying dereference safety via expanding-scope analysis. In *ISSTA '08: International Symposium on Software Testing and Analysis* (2008). to appear.
27. NANDA, M. G., GROTHOFF, C., AND CHANDRA, S. 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* (New York, NY, USA, 2005), ACM Press, pp. 77–96.
28. NAUMOVICH, G., AVRUNIN, G. S., AND CLARKE, L. A. Data flow analysis for checking properties of concurrent java programs. In *Proc. Intl. Conf. on Software Eng.* (Los Angeles, May 1999), pp. 399–410.
29. NAUMOVICH, G., CLARKE, L. A., OSTERWEIL, L. J., AND DWYER, M. B. Verification of concurrent sofware with FLAVERS. In *Proc. Intl. Conf. on Software Eng.* (May 1997), pp. 594–597.
30. RAMALINGAM, G., WARSHAVSKY, A., FIELD, J., GOYAL, D., AND SAGIV, M. Deriving specialized program analyses for certifying component-client conformance. In *Proc. ACM Conf. on Programming Language Design and Implementation* (New York, June 17–19 2002), vol. 37, 5 of *ACM SIGPLAN Notices*, ACM Press, pp. 83–94.
31. ROUNTEV, A., RYDER, B. G., AND LANDI, W. Data-flow analysis of program fragments. In *ESEC / SIGSOFT FSE* (1999), O. Nierstrasz and M. Lemoine, Eds., vol. 1687 of *Lecture Notes in Computer Science*, Springer, pp. 235–252.
32. SAGIV, M., REPS, T., AND WILHELM, R. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst. 24*, 3 (2002), 217–298.
33. SHAHAM, R., YAHAV, E., KOLODNER, E. K., AND SAGIV, S. Establishing local temporal heap safety properties with applications to compile-time memory management. In *SAS '03: 10th International Static Analysis Symposium* (2003), vol. 2694, Springer, pp. 483–503.
34. SHOHAM, S., YAHAV, E., FINK, S., AND PISTOIA, M. Static specification mining using automata-based abstractions. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis* (2007), ACM, pp. 174–184. * best paper award.
35. SHOHAM, S., YAHAV, E., FINK, S., AND PISTOIA, M. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering (TSE) 34*, 5 (Sept. 2008).
36. STEENSGAARD, B. Points-to analysis in almost linear time. In *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996* (1996), ACM, Ed., pp. 32–41. ACM order number: 549960.
37. STROM, R. E., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng. 12*, 1 (1986), 157–171.
38. WEIMER, W., AND NECULA, G. Mining temporal specifications for error detection. In *TACAS* (2005).
39. WHALEY, J., MARTIN, M., AND LAM, M. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis* (July 2002).
40. WHALEY, J., MARTIN, M. C., AND LAM, M. S. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis* (July 2002), ACM Press, pp. 218–228.
41. YANG, J., EVANS, D., BHARDWAJ, D., BHAT, T., AND DAS, M. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06: Proceeding of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ACM Press, pp. 282–291.

42. YORSH, G., YAHAV, E., AND CHANDRA, S. Generating precise and concise procedure summaries. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2008), ACM, pp. 221–234.