Effective Typestate Verification in the Presence of Aliasing

STEPHEN J. FINK and ERAN YAHAV
IBM T. J. Watson Research Center
and
NURIT DOR[1]
IBM Haifa Research Lab
and
G. RAMALINGAM[2] and EMMANUEL GEAY
IBM T. J. Watson Research Center

This paper addresses the challenge of sound typestate verification, with acceptable precision, for real-world Java programs. We present a novel framework for verification of typestate properties, including several new techniques to precisely treat aliases without undue performance costs. In particular, we present a flow-sensitive, context-sensitive, integrated verifier that utilizes a parametric abstract domain combining typestate and aliasing information. To scale to real programs without compromising precision, we present a staged verification system in which faster verifiers run as early stages which reduce the workload for later, more precise, stages.

We have evaluated our framework on a number of real Java programs, checking correct API usage for various Java standard libraries. The results show that our approach scales to hundreds of thousands of lines of code, and verifies correctness for 93% of the potential points of failure.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Algorithms, Verification

Additional Key Words and Phrases: Alias Analysis, Program Verification, Typestate

1. INTRODUCTION

In the software development lifecyle, early defect identification increases productivity by reducing development costs and improving software quality and reliability. Development organizations employ a variety of tools to identify defects early, ranging from testing and dynamic analysis to program verification techniques. One

[1]Author's current affiliation: Panaya Inc.
[2]Author's current affiliation: Microsoft Research, India.

This is a revised and extended version of a paper presented at ISSTA 2006 in Portland, Maine.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2007 ACM 0000-0000/2007/0000-0001 $5.00

class of partial program verification techniques addresses checking if programs satisfy specified safety properties (e.g. [Dwyer and Clarke 1994; Naumovich et al. 1999; Corbett et al. 2000; DeLine and Fähndrich 2001; Ball and Rajamani 2001; Foster et al. 2002; Flanagan et al. 2002; Ashcraft and Engler 2002; Ramalingam et al. 2002; Das et al. 2002; Field et al. 2003]). Defects related to these safety properties can lead to run-time failures, and may be difficult to cover with normal testing practices.

Typestate [Strom and Yemini 1986] is an elegant framework for specifying a class of temporal safety properties. In the typestate model, a finite-state automaton encodes legal program state changes, and transitions to automaton error states indicate violations of safety properties. Typestates can encode correct usage rules for many common libraries and application programming interfaces (APIs) (e.g. [Whaley et al. 2002; Alur et al. 2005]). For example, typestate can express the property that a Java program should not read data from `java.net.Socket` until the socket is connected.

This paper addresses the challenge of typestate verification, with acceptable precision, for real-world Java programs. We focus on sound verification[3]; if the verifier reports no problem, then the program is guaranteed to satisfy the desired properties. However, if the verifier reports potential problems, they may or may not indicate actual program errors. Imprecise analysis can lead a verifier to produce "false positives": reported problems that do not indicate an actual error. Users will quickly reject a verifier that produces too many false positives.

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

This paper presents several new typestate verification techniques, ranging from the simple but imprecise, to the fairly precise but somewhat expensive. We also present a staged typestate verification approach, which exploits verifiers with varying cost/precision trade-offs. Early stages employ the efficient but imprecise analyses; subsequent stages employ progressively more expensive and precise techniques. Each progressively more precise stage focuses on verifying only "parts" of the program that previous stages failed to verify.

The key technical challenge facing typestate verification for Java concerns pointer aliasing. Since all structured data in Java is heap-allocated, almost all interesting operations involve pointer dereferencing. Further, Java libraries encourage layers of encapsulation around data, which leads to multiple levels of pointer dereferencing. In order to prove that a program manipulates an object correctly, the verifier must cut through the tangle of alias relationships by which the program manipulates the object of interest.

Researchers have developed a variety of efficient flow-insensitive may-alias (pointer) analysis techniques (e.g. [Das 2000; Heintze and Tardieu 2001; Steensgaard 1996]) that scale to fairly large programs. These analyses produce a statically bounded (abstract) representation of the program's runtime heap and indicate which abstract

[3]Our implementation is sound with respect to a subset of the full Java language, excluding concurrency, reflection, and other language features discussed later.

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" [Chase et al. 1990].

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 [Cousot and Cousot 1979], a combined domain allows a more precise solution than could be obtained by solving each domain separately. 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.

1.1 Contributions

The main contributions of this paper are:

- a flow-sensitive, context-sensitive, integrated verifier that utilizes a parametric abstract domain that combines typestate and points-to abstractions.
- two new techniques to handle destructive updates, utilizing information from a preceding flow-insensitive may points-to analysis. Specifically,
    - a *uniqueness* analysis that can strengthen the results of the may points-to analysis to support "strong updates" under certain conditions, and
    - a *focus* operation, similar in spirit to the one used in shape analysis [Sagiv et al. 2002], that enables the analysis to use strong updates in certain cases.

  Though inspired by shape analysis techniques, our focus operation applies to a more efficient, abstract domain, and results in analyses that are orders of magnitude more scalable than typical shape analyses.
- a practical staged approach for sound typestate checking; the algorithm passes information from one stage to the next, in order to reduce work in latter stages.
- an empirical evaluation of the efficiency and precision of various verification techniques. The empirical results shed light on the relative importance of various techniques for treating aliases, and demonstrate the validity of a staged approach.

Our algorithms combine a preliminary flow-insensitive pointer analysis with an interprocedural abstract interpretation based on "functional" context-sensitive analysis, as defined by Sharir and Pneuli [Sharir and Pneuli 1981].

Our implementation handles the full Java language, excluding concurrency, subject to caveats described regarding dynamic language features such as reflection. The experimental results show that the staged solver verifies correctness for 93% of the potential points of failure, running in under 10 minutes across a suite of moderately-sized programs.

The rest of this paper is organized as follows: Sec. 2 provides an informal overview of the various challenges in typestate verification, and sketches our solutions. Sec. 3,

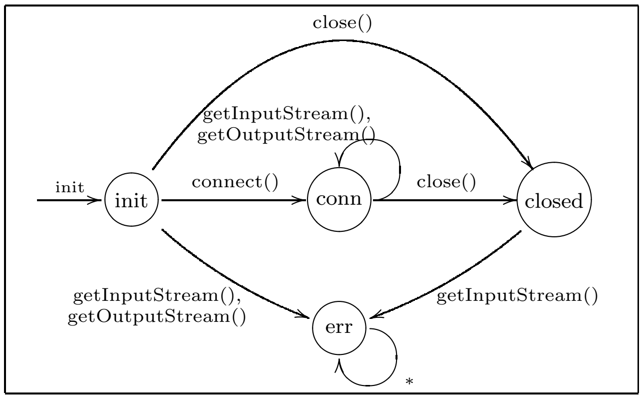


Fig. 1. Partial typestate specification for `java.net.Socket`.

Sec. 4 and Sec. 5 present the abstractions and techniques formally. Sec. 6 presents the empirical evaluation. Sec. 7 reviews related work, and Sec. 8 concludes.

2. OVERVIEW

2.1 Typestate Verification

A typestate property can be specified using a finite state automaton. States in the automaton correspond to typestates which an object can occupy during execution. The automaton also contains a designated typestate *err* corresponding to an erroneous state of the object. Transitions in the automaton correspond to *observable operations* that may change the object's typestate. In this paper, we focus on observable operations corresponding to method invocations. The goal of typestate checking is to statically verify that no object reaches its error typestate during any program execution.

Fig. 1 shows a finite state automaton providing a partial specification for the `java.net.Socket` API. This automaton shows, for example, that calling `getInputStream()` is only legal after a preceding call to `connect()`.

Fig. 2 presents a program that exercises Java Sockets, I/O streams, and Iterators. Our goal is to verify that the program

never calls `getInputStream()` or `getOutputStream()` on a `Socket` unless it is *connected*,

never calls `read()` on a *closed* stream, and

always calls `hasNext()` on an `Iterator` before calling `next()`.

In the example program, some typestate properties (e.g.Iterators) could be verified relatively easily by local, intra-procedural reasoning. Unfortunately, any local alias analysis can be easily defeated by unknown side effects from procedure calls.

Other properties require more powerful (and costly) techniques. In particular, socket usage in the example requires an interprocedural analysis with relatively precise alias analysis, since the socket objects flow across procedure boundaries and through complex collection data structures. The analysis needs to infer the

```
 class Sender {
  public static Socket createSocket() {
    return new Socket();
  }
  public static Collection createSockets() {
    Collection result = new LinkedList();
    for (int i = 0; i < 5; i++) {
      result.add(new Socket());
    }
    return result;
  }
  public static Collection readMessages() throws IOException {
    Collection result = new ArrayList();
    FileInputStream f = new FileInputStream("/tmp/foo.txt");
    // ...
    f.read();
    // ...
    return result;
  }
  public static void talk(Socket s) throws IOException {
    Collection messages = readMessages();
    PrintWriter o = new PrintWriter(s.getOutputStream(),true);
    for (Iterator it=messages.iterator();it.hasNext();) {
      Object message = it.next();
      o.print(message);
    }
    o.close();
  }
  public static void example() throws IOException {
    InetAddress ad=InetAddress.getByName("tinyurl.com/cqaje");
    Socket handShake = createSocket();
    handShake.connect(new InetSocketAddress(ad, 80));
    InputStream inp = handShake.getInputStream();

    Collection sockets = createSockets();
    for (Iterator it = sockets.iterator(); it.hasNext();) {
      Socket s = (Socket) it.next();
      s.connect(new InetSocketAddress(addr, 80));
      talk(s);
    }
    talk(handShake);
  }
}
```

Fig. 2. Program with correct usages of common APIs.

typestates of the Socket objects in the collection. Specifically, the analysis needs to determine that any Socket passed to `talk()` has been connected.

2.2 Outline of our Algorithm

Our verification system is a *composite* verifier built out of several *composable* verifiers of increasing precision and cost. Each verifier can run independently, but the composite verifier stages analyses in order to improve efficiency without compromising precision. The early stages use the faster verifiers to reduce the workload for later, more precise, stages.

All of our verifiers use the results of a preceding flow-insensitive, selectively context-sensitive subset-based pointer analysis. This analysis produces a conservative approximation of the heap, and induces a partition of concrete objects into *abstract objects*; as is typical, the pointer analysis creates names for abstract objects

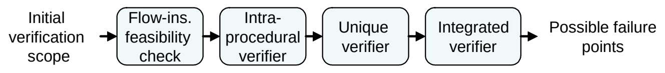

Initial verification scope
Flow-ins. feasibility check
Intra-procedural verifier
Unique verifier
Integrated verifier
Possible failure points

Fig. 3. Overview of framework stages.

based on static allocation sites and the governing context-sensitivity policy [4]. The flow-insensitive alias analysis can be performed relatively efficiently, and scales to large programs (e.g. [Heintze and Tardieu 2001; Lhoták and Hendren 2003]).

Given a program and a typestate property, we consider all operations in the program that may cause a transition to an error state as *points of potential failure (PPF).* We consider a pair (o, p) where o is an abstract object, and p a point of potential failure, as a separate verification problem. We refer to such pairs as *potential failure pairs.* We define a *verification scope* to be a set of potential failure pairs.

Our verification system starts by initializing the verification scope to contain all matching pairs of abstract typestate objects and potential points of failure. The verification scope is then gradually reduced by a sequence of stages, as shown in Fig. 3. Each stage may successfully eliminate potential failure pairs by verifying for a pair (o, p) that a failure cannot occur for objects represented by o at the point p.

Each composable verifier exploits *separation* [Das et al. 2002; Yahav and Ramalingam 2004]: it performs the typestate checking separately for each abstract object of the appropriate type in the program. It accepts as a parameter a verification scope which holds information from the preceding stages about which potential failure pairs remain unverified.

Each verifier restricts its attention to the verification scope, and produces an updated verification scope for the subsequent phase. The system reports any potential failure pairs that remain after the last stage as potential errors.

The stages in our framework are sequenced by increasing cost and precision. In the following discussion we briefly describe each of these stages. Later, we present a more detailed description of the algorithms.

2.2.1 *Flow-Insensitive Feasibility Checking.* Prior to any flow-sensitive analysis, the first stage prunes the verification scope using an extremely efficient flow-insensitive error-path feasibility check. The flow-insensitive pointer analysis provides the set of observable operations that may occur for each abstract object. The flow-insensitive verifier determines if it is possible for the abstract object to reach the *err* state in the typestate automaton, using this set of operations.

Any abstract object that does not exhibit a feasible error-path could be considered as verified.

In our running example, the `FileInputStream` object allocated in `readMessages()` is pruned at this stage, as the program never invokes `close()` for this abstract object, and thus it can never reach an error state (for "`read()` after `close()`").

This stage, however, is unable to verify the correct usage of the Iterators or Sockets in the example program.

[4] Sec. 6 gives details on our implementation's context-sensitivity policy.

2.2.2 *Intraprocedural Verifier.* The intraprocedural verifier is a flow-sensitive verifier that restricts the scope of each verification attempt to a single procedure. The verification starts at the beginning of each procedure assuming an arbitrary unknown initial context (state). Method calls are treated conservatively, without analyzing the method. This essentially works well for "local" objects, which are pointed-to by local variables only. The intraprocedural verifier uses the same abstraction as the integrated verifier (see Sec. 2.2.4 and Sec. 5).

When the intraprocedural verifier is able to verify all uses of an abstract object in the program, we can avoid interprocedural verification for that object. This is often the case for typestate objects that do not escape the method in which they are allocated.

For example, the intraprocedural verifier can verify that all Iterators in our running examples are used correctly. Applying the intraprocedural verifier as an early stage eliminates the need for verification of the Iterators in the running example by the latter, more expensive interprocedural solvers.

2.2.3 *Strong Updates: Uniqueness Analysis.* While a flow-insensitive alias analysis suffices to check feasibility of an error-path (as in Sec. 2.2.1), it generally does not suffice for verifying typestate properties. A flow-insensitive analysis produces only *may alias* information and not *must alias* information. Therefore, an analyzer that directly uses the results of a flow-insensitive analysis must use "weak updates" to handle assignments and operations via a pointer.

Using "weak updates" precludes verification of many typestate properties. For example, it is insufficient for verifying the typestate property of Fig. 1. Using only *may alias* information, the analyzer cannot guarantee that a `connect()` operation occurs on the same concrete object as a subsequent `getInputStream()` operation. Hence, such analysis cannot verify this property.

We present a verifier based on flow-insensitive alias analysis, but that uses a novel *uniqueness analysis* to allow strong updates in some scenarios. Uniqueness analysis strengthens the information obtained via flow-insensitive may points-to analysis by identifying *unique* abstract objects that represent a single concrete object at a given point of the program. For the purpose of typestate checking, a pointer that may-point to a single target which is a unique abstract object may be assumed to must-point to that abstract object.

Consider the invocation of a method, via a pointer p, that may alter the typestate of the receiver object. If the following two conditions hold, then the analysis can apply a strong update to change the typestate of the receiver object:

(a) the points-to set for p consists of a single *abstract* object

(b) this *abstract* object represents a single *concrete* object[5].

Consider an abstract object S representing a particular (context-sensitive) allocation site A. This abstract object represents all concrete objects that are allocated at A. The Unique solver performs a flow- and context-sensitive analysis with a simple

[5]For purposes of typestate checking, we may safely ignore the possibility of the pointer `p` being null, which will result in a null-pointer dereference exception. If desired, null-pointer checking is done separately.

abstraction to determine if more than one object allocated at A can be simultaneously alive. If not, then the abstract object S represents at most one concrete object at that program point, and the verifier can exploit strong updates at that point if condition (a) mentioned above also holds.

For example, the Unique verifier can verify the correct use of the socket pointed-to by `handShake` in the method `example()`, despite the fact that this object is used interprocedurally (and hence could not be handled by the intraprocedural verifier of the previous section).

Uniqueness analysis is of general use in our framework, and later stages incorporate the technique. This novel analysis compares favorably to existing techniques for computing unique abstract locations, as it relies on flow- and context-sensitive analysis of a pruned program with respect to the tracked abstract object (see discussion in Sec. 7).

2.2.4 *Integrated Verifier.* The Integrated verifier improves upon the Unique verifier by performing flow- and context-sensitive verification with an abstraction that combines aliasing information with typestate information. The use of a combined domain is more precise than separately performing typestate checking and flow-sensitive alias analysis, as is common with abstract interpretation over combined domains [Cousot and Cousot 1979].

For example, flow-sensitivity of alias information enables strong-updates in cases such as the one below, where the Unique verifier fails because the abstract file object does not qualify as unique.

```
Collection files = ...
while (...) {
  File f = new File();
  files.add(f);
  f.open();
  f.read();
}
```

Since all our verifiers exploit separation, it suffices to focus on the problem of verifying usage for a single abstract object. The Integrated verifier utilizes an abstract domain that captures information about the typestate of the given abstract object, as well as information about a set M of pointer access paths that definitely point to the given abstract object, and a set MN of pointer access paths that definitely do not point to the given abstract object. The domain also includes a boolean flag indicating if there may exist other access paths, not mentioned in M, that may point to the given abstract object. Sec. 5 presents a more complete description of the abstraction.

A key element of the integrated verifier's abstraction is the use of a *focus* operation [Sagiv et al. 2002], which is used to dynamically (during analysis) make distinctions between objects that the underlying basic points-to analysis does not distinguish. For example, consider the loop in the method `example()` in our running example. The verifier utilizes two or more abstract objects to represent the set of all (5) `Socket` objects created by the `createSockets()` method (even though the flow-insensitive pointer analysis represents them by a single abstract object):

one abstract object represents the Socket pointed to by `s`, and the other abstract objects represent the remaining Sockets.

This enables the use of strong updates, allowing verification for all Sockets in the running example, despite their flow through a collection and across procedures.

3. TYPESTATE CHECKING FRAMEWORK

This section presents a framework for typestate checking which enables declaration of different levels of abstractions.

First, we sketch an instrumented concrete semantics for this problem. Intuitively, given a typestate property, our semantics instruments the program state, $state^\natural$ to include for every object, $o^\natural$, its typestate from the property definition. The instrumented semantics verifies that an object never reaches its error typestate.

Next, we present a *parameterized* conservative abstraction that allows us to define the family of abstractions used by the various verifiers in our framework.

3.1 Instrumented Concrete Semantics

We assume a standard concrete semantics which defines a program state and evaluation of an expression in a program state. The semantic domains are defined in a standard way as follows:

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

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 (instance) field identifiers.

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

We also define the notion of an access path as follows: A *pointer path* $\gamma \in \Gamma = FieldId^*$ is a (possibly empty) sequence of field identifiers. The empty sequence is denoted by ϵ. We use the shorthand f^k where $f \in FieldId$ to mean a sequence of length k of accesses along a field f. An *access path* $p \equiv x.\gamma \in VarId \times \Gamma$ is a pair consisting of a local variable x and a pointer path γ.

We denote by APs all possible access paths in a program. The r-value of access path p, denoted by $state^\natural[p]$, is recursively defined using the environment and heap mappings, in the standard manner. Intuitively, for an access path p, $state^\natural[p]$ is the set of objects to which p refers.

We formally define a typestate property as follows.

DEFINITION 3.1. *A typestate property $\mathcal{F}$ is represented by a finite state automaton $\mathcal{F} = \langle \Sigma, \mathcal{Q}, \delta, init, \mathcal{Q} \setminus \{err\} \rangle$ where Σ is the alphabet of observable operations, $\mathcal{Q}$ is the set of states, δ is the transition function mapping a state and an operation to a successor state, $init \in \mathcal{Q}$ is a distinguished* initial state*, $err \in \mathcal{Q}$ is a distinguished* error state *for which for every $\sigma \in \Sigma$, $\delta(err, \sigma) = err$, and all states in $\mathcal{Q} \setminus \{err\}$*

are accepting states. Given a sequence of operations we say that it is valid *when it is accepted by $\mathcal{F}$, and* invalid *otherwise.*

Our instrumented concrete semantics instruments every concrete state $\langle L^\natural, \rho^\natural, h^\natural \rangle$ with an additional mapping $typestate^\natural : L^\natural \to \mathcal{Q}$ that maps an allocated object to its typestate.

For a given state $state^\natural = \langle L^\natural, \rho^\natural, h^\natural \rangle$, we define a function $AP^\natural_{state^\natural} : L^\natural \to 2^{APs}$ as a mapping between allocated objects and the access paths that evaluate to them, i.e. $AP^\natural(o^\natural) = \{e \mid state^\natural[e] = o^\natural\}$. When the state is clear from context, we omit it and simply write $AP^\natural(o^\natural)$.

A state of the instrumented concrete semantics is therefore a tuple $\langle L^\natural, \rho^\natural, h^\natural, typestate^\natural \rangle$.

EXAMPLE 3.2. *Given the property of Fig. 1, the instrumented concrete state before the first call to* `s.connect()` *in* `example()` *contains six objects: one object $o^\natural_0$ allocated during the invocation of* `createSocket()`*, and five other objects $o^\natural_1, \dots, o^\natural_5$, allocated during the invocation* `createSockets()`*. The values of typestate$^\natural$ and the function $AP^\natural(o^\natural_0)$ are:*

$$\begin{array}{ll}
typestate^\natural(o^\natural_0) = conn & AP^\natural(o^\natural_0) = \{\texttt{handShake}\} \\
typestate^\natural(o^\natural_1) = init & AP^\natural(o^\natural_1) = \{\texttt{s}, sockets.head\} \\
typestate^\natural(o^\natural_i) = init & AP^\natural(o^\natural_i) = \{sockets.head.next^{i-1}\} \\
\quad where(i = 2, ..5) &
\end{array}$$

The instrumented semantics updates the typestate of the object in a natural way. When the object is first allocated, its typestate is mapped to the initial state of the typestate automaton. Then, on every observable event, the object typestate is updated accordingly.

In this paper, we consider method invocations as observable events which can change typestate. We use an instrumented semantics where the typestate changes during the method call. When the first instruction of the called method executes, the object has already transitioned to the appropriate typestate. If the called method in turn calls other methods that cause typestate transitions, the instrumented semantics tracks these transitions as well.

3.2 Abstract Semantics

The instrumented concrete semantics uses an unbounded set of objects with an unbounded set of (unbounded) access paths. In this section, we describe a parameterized abstract semantics that allows us to conservatively represent the instrumented concrete semantics with various degrees of precision and cost.

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.

3.2.1 *Flow-insensitive May Points-to 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 [Andersen 1994]. This is fairly standard and provides a partition of the set *objects*$^\natural$ 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.

Our analysis framework can work with any pointer analysis that provides a points-to solution mapping abstract pointers to abstract objects. Of course, there exist many variants of flow-insensitive pointer analysis depending on context-sensitivity policies [Grove and Chambers 2001]. Section 6 describes the pointer analysis variant used in our current implementation.

The heap graph representation of the running example contains two instance keys for type Socket: one representing the object allocated in `createSocket`, denoted by $o_0^\natural$ in Example 3.2, and another one, for the second allocation site, representing all five objects in the `sockets` collection.

3.2.2 *Parameterized Typestate Abstraction.* Our parameterized abstract representation uses tuples of the form: $\langle o, \mathit{unique}, \mathit{typestate}, AP_{\mathit{must}}, \mathit{May}, AP_{\mathit{mustNot}} \rangle$ 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 only 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 $\mathit{May} = \mathit{false}$, the must access path is complete; it contains all access paths to this object.

(e) If an object occupies a typestate T, the corresponding abstract state must also indicate the same typestate, T.

This can be formally stated as follows:

Definition 3.3. *A tuple* $\langle o, \mathit{unique}, \mathit{typestate}, AP_{\mathit{must}}, \mathit{May}, AP_{\mathit{mustNot}}\rangle$ *is a* sound representation *of object* $o^\natural$ *at instrumented state* $\mathit{istate}^\natural$ *when:*

$$\begin{aligned}
&o = \mathit{ik}(o^\natural)\\
&\ \wedge \mathit{unique} \Rightarrow \{x^\natural \in \mathit{live}(\mathit{istate}^\natural) \mid \mathit{ik}(x^\natural) = o\} = \{o^\natural\}\\
&\ \wedge \mathit{typestate} = \mathit{typestate}^\natural(o^\natural)\ \wedge AP_{\mathit{must}} \subseteq AP^\natural(o^\natural)\\
&\ \wedge (\neg \mathit{May} \Rightarrow (AP_{\mathit{must}} = AP^\natural(o^\natural)))\\
&\ \wedge AP_{\mathit{mustNot}} \cap AP^\natural(o^\natural) = \emptyset
\end{aligned}$$

where ik is an abstraction mapping a concrete object to the instance key that represents it, and $\mathit{live}(\mathit{istate}^\natural)$ *is defined to be* $\{x^\natural \mid AP^\natural(x^\natural) \neq \emptyset\}$.

Note that the set $\mathit{live}(\mathit{istate}^\natural)$ in the above definition identifies the set of all non-garbage objects in the state $\mathit{istate}^\natural$, i.e. the set of all objects reachable via at least one access path. As we explain in Sec. 4, our implementation is actually based on a refined definition, where $\mathit{live}(\mathit{istate}^\natural)$ corresponds to the set of all objects reachable via at least one *live* access path (i.e., an access path that may be used in the future). This allows us to ignore reachable objects that will not be used in the future and can be treated as garbage.

Definition 3.4. *An abstract state istate is a* sound *representation of a concrete state* $\mathit{istate}^\natural = \langle L^\natural, \rho^\natural, h^\natural, \mathit{typestate}^\natural\rangle$ *if for every object* $o^\natural \in L^\natural$ *there exists a tuple in istate that provides a sound representation of* $o^\natural$.

3.3 Base Abstraction

The Base (least precise) abstraction is an instance of the parameterized abstraction with zero length and width, for both the must and the must-not access path sets (and hence $\mathit{May} = \mathit{true}$ in all tuples). In addition, this abstraction does not track uniqueness. This yields a typestate checking algorithm, similar to [Das et al. 2002] in its alias handling, that cannot verify any property that requires strong updates. For simplicity, we denote each tuple in this abstraction as $\langle o, \mathit{typestate}\rangle$

Example 3.5. *A base abstraction representing the concrete state described in Example 3.2 contains two instance keys:* o_0 *representing* $o_0^\natural$ *and* $o_{1..5}$ *representing the five objects* $o_i^\natural, i = 1, 2, ..5$ *in the* `sockets` *collection and the following three tuples:* $\langle o_0, \mathit{init}\rangle$, $\langle o_0, \mathit{conn}\rangle$, $\langle o_{1..5}, \mathit{init}\rangle$.

This analysis is an iterative flow- and context-sensitive propagation, that tracks tuples starting with an initial $\langle o, \mathit{init}\rangle$ generated at an allocation. The analysis only needs to handle observable operations and propagates tuples according to typestate changes. The result of an observable operation associated with event *op* on the tuple $\langle o, \mathit{typestate}\rangle$ are two tuples: The previous tuple and the tuple $\langle o, \delta(\mathit{typestate}, \texttt{op})\rangle$. Tuples are never removed; all operations are handled as weak updates. The first tuple in Example 3.5 demonstrates the results of a weak-update. It represents that $o_0^\natural$, in Example 3.2, may be in the *init* state, which is not feasible in any concrete state at this program point.

4. UNIQUENESS ANALYSIS

The Unique verifier extends the Base abstraction, adding an abstraction which determines whether more than one concrete object corresponding to a given instance key can be simultaneously alive. This information allows the verifier to use strong updates under certain conditions. We refer to this analysis as *uniqueness analysis*.

In terms of the abstraction tuples introduced in Sec. 3, the Unique verifier makes use of only the instance key, uniqueness flag, and the typestate. (Thus the must-point-to set and must-not-point-to set are always empty, and the May flag is always true.) Hence, we will represent each tuple as a triple $\langle o, \mathit{unique}, \mathit{typestate}\rangle$.

The analysis works as follows. The first time an allocation site with an instance key k is executed (during analysis), it generates the tuple $\langle k, \mathit{true}, \mathit{init}\rangle$. If, during the analysis, any tuple $\langle k, \mathit{true}, \mathit{typestate}\rangle$ reaches the same (context-sensitive) allocation site, the allocation site will generate the tuple $\langle k, false, \mathit{typestate}\rangle$.

Note that our notion of a *unique object* differs from the notion of a *unique pointer* as it often appears in the literature [Crary et al. 1999]. Our unique object corresponds to a single live object created at a particular allocation site, although there may exist multiple pointers to it. In contrast, a unique pointer usually denotes a pointer which does not alias any other pointer within a particular scope.

To make the above technique effective for allocation sites that are in a loop, it is necessary to find a way to "kill" the tuples where possible. This verifier utilizes a preliminary liveness analysis, computed prior to typestate checking, that determines a conservative approximation of which instance keys may be live at each program point. Whenever a tuple p for an instance key o flows to a program point where o cannot be live, p can be removed soundly.

The framework admits any form of liveness analysis, which can be plugged into the verifier. Our current implementation uses a simple bottom-up interprocedural liveness analysis, based on the results of the preliminary flow-insensitive, partially context-sensitive pointer analysis.

This approach is effective in two situations. First, *singleton* pattern objects clearly retain their *unique* predicates, and so enjoy strong updates everywhere. The Java standard libraries use singleton patterns frequently. For example, consider the `EMPTY_SET.iterator()` object which appears in the Java Collections framework. This singleton object tends to flow everywhere in the program that uses collections, defeating the pruning optimizations described earlier. However, the *unique* predicate allows the solvers to efficiently treat this object with strong updates, without resorting to expensive access-path tracking.

Additionally, the liveness analysis allows unique analysis to succeed for a ubiquitous pattern: an allocated object dies before its allocation site executes again. In practice, we have found that a simple liveness analysis catches many of these cases.

For tuples not marked unique, this verifier degenerates into the Base verifier of Sec. 3.3. For example, while uniqueness handles the *handshake* socket in the running example, uniqueness cannot show that the Sockets in the collection are used correctly. The instance key that represents all the Socket objects in the `sockets` collection is, naturally, not unique. Therefore, when the statement `s.connect()` is analyzed, the typestate of the abstract Socket object is weakly-updated, indicating that a socket may occupy the *conn* state or the *init* state. These tuples propagate to the statement `s.getOutputStream()` in `talk()`, causing the verifier to imprecisely report a possible error.

Note that in the example, although verifying usage of the *handshake* object does not rule out errors at many potential points of failure, the staged verifier will remove pairs involving the *handshake* object from the running verification scope. This would reduce the computational workload for the next stage.

5. INTEGRATED TYPESTATE AND ALIAS ANALYSIS

In this section, we describe two verifiers that make use of the access-path sets in the tuple representation.

We first describe the APFocus verifier, our most precise analysis. This analysis uses the three remaining tuple components, AP_{must}, May, and $AP_{mustNot}$, to track alias relationships precisely with flow- and context-sensitive analysis. With this more precise information, the solver can potentially apply strong updates based on must-point-to information and avoid spurious updates based on must-not-point-to information. Furthermore, we describe a merge operator to avoid redundant work, and discuss how the abstractions give a measure of path sensitivity for common scenarios involving virtual method dispatch.

We begin by presenting the flow functions for the APFocus verifier.

5.1 Update Functions

The APFocus verifier updates the tuple state when propagating information through statements that may affect tracked alias or typestate relations. Table I shows how a tuple is transformed by the interpretation of various statements.

The interpretation of an allocation statement "`v = new T()`" with instance key o will *generate* a tuple $\langle o, true, init, \{v\}, false, \emptyset\rangle$ representing the newly allocated object. When May is false, the $AP_{mustNot}$ component is redundant and, hence, initialized to be empty.

When a typestate method is invoked, we can (1) use the $AP_{mustNot}$ information to avoid changing the typestate of the tuple where possible, (2) use the AP_{must} information to perform strong updates on the tuple where possible, and (3) use the uniqueness information also to perform strong updates where possible.

When a tuple reaches the allocation site that created it, we generate two tuples, one representing the newly created object, and one representing the incoming tuple. We change the uniqueness flag to false for reasons explained earlier. For assignment statements, we update the AP_{must} and $AP_{mustNot}$ as appropriate.

Note that since we place a finite bound on access path lengths, there are a finite number of possible abstract states, so fixed-point iteration terminates. The number of possible abstract states is exponential in the access path bound. The flow functions are distributive since they propagate only one tuple at a time.

Stmt S	**Resulting abstract tuples**
observable operation $e.op()$ as $\texttt{op} \in \Sigma$ where $o \in pt(e)$	$\langle o, unique, \delta(typestate, \texttt{op}), AP_{must}, May, AP_{mustNot} \rangle$ if $e \notin AP_{mustNot} \wedge (e \in AP_{must} \vee May)$ $\langle o, unique, typestate, AP_{must}, May, AP_{mustNot} \rangle$ if $e \in AP_{mustNot} \vee (e \notin AP_{must} \wedge \neg(unique \wedge pt(e) = \{o\}) \wedge May)$
$v = \texttt{new}\ \mathrm{T}()$ where $o =$ Stmt S	$\langle o, false, typestate, AP_{must} \setminus \{v.\gamma \mid \gamma \in \Gamma\}, May, AP_{mustNot} \cup \{v\} \rangle$ $\langle o, false, init, \{v\}, false, \emptyset \rangle$
$v = \texttt{null}$	$\langle o, unique, typestate, AP_{must} \setminus \{v.\gamma \mid \gamma \in \Gamma\}, May, AP_{mustNot} \cup \{v\} \rangle$
$v.f = \texttt{null}$	$\langle o, unique, typestate, AP_{must} \setminus \{e'.f.\gamma \mid mayAlias(e', v), \gamma \in \Gamma\}, May, AP_{mustNot} \cup \{v.f\} \rangle$
$v = e$	$\langle o, unique, typestate, AP_{must} \cup \{v.\gamma \mid e.\gamma \in AP_{must}\}, May, AP_{mustNot} \setminus \{v \mid e \notin AP_{mustNot}\} \rangle$
$v.f = e$	$AP'_{must} := AP_{must} \cup \{v.f.\gamma \mid e.\gamma \in AP_{must}\}$ $\langle o, unique, typestate, AP'_{must}, May \vee \exists v.f.\gamma \in AP'_{must}.\exists p \in AP.mayAlias(v, p) \wedge p.f.\gamma \notin AP'_{must}, AP_{mustNot} \setminus \{v.f \mid e \notin AP_{mustNot}\} \rangle$

Table I. Transfer functions for statements indicating how an incoming tuple $\langle o, unique, typestate, AP_{must}, May, AP_{mustNot} \rangle$ is transformed, where $pt(e)$ is the set of instance keys pointed-to by e in the flow-insensitive solution, $v \in VarId$. $mayAlias(e_1, e_2)$ iff pointer analysis indicates e_1 and e_2 may point to the same instance key.

5.2 Focus Operation

We now describe the focus operation, which improves the precision of the analysis. As a motivating example, consider the statement `s.connect()` in the loop in the method `example()` in our running example. 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 `s` points to, and another tuple to represent the remaining sockets. Formally, consider an incoming tuple $\langle o, \mathit{unique}, \mathit{typestate}, AP_{\mathit{must}}, \mathit{true}, AP_{\mathit{mustNot}} \rangle$ at an observable operation $e.\mathit{op}()$, where $e \notin AP_{\mathit{must}}$, but e may point to o (according to the flow-insensitive points-to solution). The analysis replaces this tuple by the following two tuples:

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

In the example under consideration, the statement `s.connect()` is reached by the tuple $\langle o_{1..5}, false, \mathit{init}, \emptyset, \mathit{true}, \emptyset \rangle$. Focusing replaces this tuple by the following two tuples:

$$\langle o_{1..5}, false, \mathit{init}, \{s\}, \mathit{true}, \emptyset \rangle$$
$$\langle o_{1..5}, false, \mathit{init}, \emptyset, \mathit{true}, \{s\} \rangle$$

The invocation of `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:

$$\langle o_{1..5}, false, \mathit{conn}, \{s\}, \mathit{true}, \emptyset \rangle$$
$$\langle o_{1..5}, false, \mathit{init}, \emptyset, \mathit{true}, \{s\} \rangle$$

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 $\langle o_{1..5}, false, \mathit{conn}, \{s\}, \mathit{true}, \emptyset \rangle$, however, represents a single object at this point, namely the object pointed to by s, 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.

5.3 Focus and polymorphism

Polymorphism is the distinguishing feature of object-oriented languages; an object's behavior depends on its concrete type rather than it's declared type. Polymorphic call sites present an interesting and widespread difficulty for the integrated typestate checking.

Consider the following snippet of code:

```
Collection c = ...
for (Iterator it=c.iterator(); it.hasNext();){
  it.next();
}
```

The Java Collections API often returns one of two Iterator implementations, depending on whether the collection is empty. Thus, the calls to both `hasNext` and `next` are polymorphic. This effectively introduces a path-sensitivity issue, where the two dynamic dispatch sites play the role of correlated branches in traditional path-sensitive discussions.

As in ESP [Das et al. 2002], we could introduce path-sensitive predicates that encode the direction of dynamic dispatch. Instead, our focus algorithms exploit information from the tuple to avoid propagation at polymorphic call sites.

In particular, before the call to `hasNext`, if we have the tuple $\langle o, false, init, \emptyset, true, \emptyset \rangle$ (in which case o represents one of the two possible concrete `Iterator` implementations) then the *focus* operation will result in two tuples after the call to `hasNext`:

$$t_1 = \langle o, false, hasNext, \{\texttt{it}\}, true, \emptyset \rangle$$
$$t_2 = \langle o, false, init, \emptyset, true, \{\texttt{it}\} \rangle$$

The flow functions for *call* edges exploit alias information to avoid propagating tuples down infeasible paths. In particular, the flow function for the call to `it.next` will not propagate t_2 to the `next` operation, since t_2 indicates that `it` *must-not* alias o. Thus, focus avoids a spurious transition to *err*.

Intuitively, *focus* introduces a notion of path-sensitivity, where a path corresponds to a dynamic dispatch governed by alias relationships for tracked objects.

5.4 Discarding Access Paths

As explained earlier, we enforce limits on the length and the number of access paths allowed in the AP_{must} and $AP_{mustNot}$ components to keep the number of tuples generated finite. We designed the abstract domain specifically to discard access-path information soundly, allowing heuristics that trade precision for performance but do not sacrifice soundness. This feature is crucial for scalability; the analysis would suffer an unreasonable explosion of dataflow facts if it soundly tracked every possible access path, as in much prior work [Dor et al. 2004; Landi and Ryder 1992; Choi et al. 1993; Emami et al. 1994].

We can always safely discard access path elements from the $AP_{mustNot}$ component, since the flow functions do not rely on the must-not set being complete. Additionally, we can safely discard elements from the AP_{must} component by setting the *May* component to be true, indicating that the AP_{must} set does not contain all possible aliases.

There are a variety of possible heuristic options for limiting the number of tuples. For example, ESP's "property simulation" introduced lossy joins, to merge tuples that do not differ in the typestate property of interest [Das et al. 2002].

Our current implementation uses a different heuristic. It discards the prior $AP_{mustNot}$ paths when applying a *focus* operation, maintaining the more precise information from the most recent *focus*. This is based on intuition that in most cases the extra precision from *focus* will manifest at the next typestate change. This heuristic avoids a common exponential blowup in state due to a sequence of focus operations, and seems to perform well in practice.

5.5 The APMust Verifier

APMust is a simpler version of APFocus engine that makes use of the AP_{must} component, but not the $AP_{mustNot}$ component. Thus, the $AP_{mustNot}$ component is always an empty set in this abstraction. Since it does not use the $AP_{mustNot}$,

```
 foo(Collection c) {
   FileComponent f = new FileComponent(); // AllocSite1
   c.add(f);
   if (?)  {
      x1 = f;
   }
   if (?)  {
      x2 = f;
   }
   if (?)  {
      x3 = f;
   }
   ...
   if (?)  {
      xk = f;
   }
   (1) ...
}
```

Fig. 4. Example that manifests an exponential blowup due to must-path factoids without normalization.

it does not use focus either (since focus is ineffective without the $AP_{mustNot}$). Other aspects of this engine, such as the transfer functions, can be obtained in a straightforward way from the description of APFocus.

We include the APMust verifier for comparison in the next section, to help evaluate the contribution of the focus operation.

5.6 Join Operation

The RHS tabulation algorithm ([Reps et al. 1995]) assumes that dataflow facts in the domain are independent, and uses set union as the join operation. Our abstract domain can be treated similarly, propagating each tuple independently. However, this approach can lead to an exponential blowup of work, as typical in path-sensitive analyses.

Consider the example in Figure 4. In this example, at program point (1), the must-path abstraction will maintain tuples with must-sets containing all subsets of {x1,. . . , xk}. The aliasing relationship effectively encodes each of the possible 2^k paths through this method.

We therefore use a slight generalization of the RHS algorithm which supports join operations other than set union. Using such join operations potentially allows us to sacrifice the precision guaranteed by the original algorithm in favor of scalability. Additionally, we show that we can apply a join operator *without* losing precision.

Observe that our abstractions induce a partial order over tuples, where one tuple may represent strictly more precise information than another.

Definition 5.1. *(Tuple partial order) Consider two tuples*

$$t_1 = \langle o^1, \mathit{unique}^1, \mathit{typestate}^1, AP^1_{\mathit{must}}, \mathit{May}^1, AP^1_{\mathit{mustNot}} \rangle$$
$$t_2 = \langle o^2, \mathit{unique}^2, \mathit{typestate}^2, AP^2_{\mathit{must}}, \mathit{May}^2, AP^2_{\mathit{mustNot}} \rangle$$

We say that t_1 is more precise than t_2 and write $t_1 \preceq t_2$ when the following conditions hold:

$o^1 = o^2$

$\mathit{typestate}^1 = \mathit{typestate}^2$

May^2

$\neg \mathit{unique}^1 \vee \mathit{unique}^2$

$AP^1_{\mathit{must}} \supseteq AP^2_{\mathit{must}}$

$AP^1_{\mathit{mustNot}} \supseteq AP^2_{\mathit{mustNot}}.$

It is easy to see that if all these conditions hold, each component of tuple t_1 holds the same or more precise information than the corresponding component of t_2. So, any program states represented by t_1 are also redundantly represented by t_2. If both t_1 and t_2 reach the same program point (in the same context), there is no additional precision gained in propagating tuple t_1, since the less precise tuple t_2 covers these cases and represents a lower bound on the eventual least fixed-point.

As an extreme example, consider the tuple $\langle \mathit{AllocSite}1, \mathit{false}, \mathit{init}, \emptyset, \mathit{true}, \emptyset \rangle$. This tuple represents the least precise must and must-not information (empty sets of access paths), the least precise may information (may bit is true) and least precise unique information (unique bit is false).

Note that if $\mathit{May}^2 = \mathit{false}$, then the AP^2_{must} set holds exactly the set of expressions which name the corresponding object and typestate. There is no other tuple with more precise information than t_2.

In the example of Fig. 4, note that the following three tuples (among others) will propagate to program point (1):

$$t_1 = \langle \mathit{AllocSite}1, \mathit{true}, \mathit{init}, \{\mathtt{x1}\}, \mathit{true}, \emptyset \rangle$$
$$t_2 = \langle \mathit{AllocSite}1, \mathit{true}, \mathit{init}, \{\mathtt{x2}\}, \mathit{true}, \emptyset \rangle$$
$$t_3 = \langle \mathit{AllocSite}1, \mathit{true}, \mathit{init}, \{\mathtt{x1}, \mathtt{x2}\}, \mathit{true}, \emptyset \rangle$$

Since $t_3 \preceq t_1$, we may propagate t_1 and kill the (more precise) tuple t_3. This does not lose precision compared to propagating all three tuples, since the final least fixed point will be bounded by the precision of the less precise tuples t_1 and t_2.

Alternatively, we could instead introduce join operators which sacrifice precision to accelerate convergence. For example, the ESP system [Das et al. 2002] presented *property simulation*, which aggressively merges facts associated with a particular typestate. In our abstractions, an analogous strategy could merge the access-path information regarding all facts describing a particular abstract object in a particular typestate.

Note that in our abstractions, these redundant facts can arise from any statement that modifies aliases, and not just at control-flow merge points. Our solver applies the merge operator at every program point.

The partial order over tuples induces a Hoare order over abstract states. Given two abstract states a_1 and a_2, we say that a_1 is more precise than a_2, and write $a_1 \sqsubseteq a_2$, when for every tuple t_1^i in a_1 there exists a tuple t_2^j in a_2 such that $t_1^i \preceq t_2^j$. Our framework can be instantiated with any sound join operation, that is, with any join operation $\sqcup$ that guarantees $a_1 \cup a_2 \sqsubseteq a_1 \sqcup a_2$.

5.7 Using the Structure of the Verified Property

The verifiers we described in previous sections are all sound for any arbitrary typestate property. In this section, we consider properties of a certain structure, and show that they can be checked using a specialized, more efficient, verifier.

The general idea is to exploit the nature of the property being verified for devising a more efficient verifier. We now consider the class of *omission closed* typestate properties, originally introduced in [Field et al. 2003].

Informally, a property is *omission closed* if the set of sequences accepted by the property automaton is closed with respect to omissions: any sequence obtained by omitting one or more operations from an accepted sequence of operations is also accepted.

DEFINITION 5.2. *A property represented by an automaton $\mathcal{F}$ is said to be* omission-closed *when for all sequences $\alpha, \beta, \gamma \in \Sigma^*$, if α, β, γ is valid then $\alpha\gamma$ is also valid.*

We observe that for this class of properties, it is sound to perform a strong-update based only on may points-to information, without considering must points-to information. This provides a sound verifier that permits strong updates without maintaining any must points-to information.

The intuition is that for omission closed properties, assuming a strong-update is actually the worst-case assumption. Any error sequence that is present when using weak updates is also present when using strong updates.

Omission closed properties have a finite set of *forbidden subsequences* $\xi_1, \xi_2, \ldots, \xi_k$ such that a sequence α is invalid iff α contains some ξ_i as a subsequence.

This, together with the fact that our abstraction represents objects at different states by separate tuples, guarantees that we will observe any invalid sequence by using strong-updates.

Any sequence that is found using weak-updates has a sub-sequence that only uses strong-updates. Therefore, if there is an invalid sequence using weak-updates, it has an invalid sub-sequence that only uses strong updates.

Note that for other classes of properties, such as *repeatable enabling sequence* ([Field et al. 2003]) properties, a verifier that only uses strong-updates is unsound.

EXAMPLE 5.3. *Consider the property that requires that an `InputStream` is not read from after it has been closed. Assume that the property automaton has an alphabet $\Sigma = \{read, close\}$ and that the valid sequences of events can be characterized by the regular expression `read`*; `close`.*

This property is omission closed, as any valid sequence of calls remains valid even when some calls are omitted from it (i.e., for any valid sequence, it is possible to omit either a `close` or a `read` and maintain its validity).

Consider the program of Fig. 5. Since we know that the verified property is omission-closed, we can use a solver that employs strong-updates based on may-

```
InputStream x1 = new InputStream(...); // AllocSite1     {⟨AllocSite1, init⟩}
if (?)
   x2 = x1;
x1.read();                                               {⟨AllocSite1, init⟩}
x1.close();                         {⟨AllocSite1, init⟩, ⟨AllocSite1, closed⟩}
x2.read();                           {⟨AllocSite1, closed⟩, ⟨AllocSite1, err⟩}
```

Fig. 5. Example program demonstrating the use of strong-updates for omission closed properties. Figure shows the abstract state after every typestate-relevant statement when using weak-updates. Note that the invalid sequence would have also been observed when using only strong updates.

information. This means that after the call to `close` *in the program, we will only maintain the state* $\langle AllocSite1, closed\rangle$ *instead of the set*

$$\langle AllocSite1, closed\rangle, \langle AllocSite1, init\rangle$$

Note that this still allows us to observe that the call to `x2.read` *leads to an error state.*

6. EXPERIMENTAL RESULTS

6.1 Implementation

The preliminary flow-insensitive pointer analysis provides a mostly context-insensitive field-sensitive Andersen's analysis [Andersen 1994], enhanced with a selective object sensitivity policy [Milanova et al. 2005]. The pointer analysis relies on an SSA register-transfer language representation of each method, which gives a measure of flow-sensitivity for points-to sets of local variables [Hasti and Horwitz 1998]. The pointer analysis names each context-sensitive allocation site as an instance key, and builds the call graph on-the-fly.

The pointer analysis uses a mostly context-insensitive policy; most methods are represented by exactly one context. However, some methods are cloned to disambiguate pointer flow in specific contexts. Most significantly, Java collection classes and I/O stream containers are treated with unlimited depth (up to recursion) object-sensitivity [Milanova et al. 2005]. That is, for each allocation site of a Java collection or I/O stream, the analysis creates a context (clone) for each method of the class, representing said method when invoked on an object from that particular allocation site. Furthermore, the analysis recursively applies object-sensitive cloning to all objects allocated in such methods. As a result, the contents of Java collections from different allocation sites are fully disambiguated, eliminating a major source of pointer analysis pollution.

Additionally, the pointer analysis adds one-level of call-string context to calls to a few library factory methods, including `arraycopy`, and `clone` statements. These methods tend to badly pollute pointer flow precision if handled without context-sensitivity.

The analysis deals with reflection by tracking objects to casts, as in [Fink et al. 2004; Livshits et al. 2005] . When an object is created by a reflective call (e.g. `Class.newInstance` or `Constructor.newInstance`), the analysis assumes (unsoundly) that the object will be cast to a declared type before being accessed. The analysis tracks these flows, and infers the type of object created by `newInstance` based on the declared type of relevant casts. While technically unsound, we believe that this approximation is accurate for the vast majority of reflective factory methods in Java programs. Our current implementation does not resolve calls via `Method.invoke` nor field accesses through `java.lang.reflect.Field`. In general, precise but conservative treatment of reflection remains a difficult challenge, beyond the scope of this work.

The system uses a substantial library of models of native code behavior for the standard libraries. We have built up these native method models over years of development of the analysis infrastructure in various products. We have no guarantee that our native method models are complete or adequate. A systematic way to model native methods or deal soundly with unmodelled methods remains a challenge, beyond the scope of this work.

For these experiments, we configure the analysis to ignore some system libraries such as `java.awt` and `javax.swing`. In our current implementation, if these libraries are included in the analysis scope, the treatment of reflection causes the computed call graph to blow up immensely, beyond our ability to scale the typestate verification. We believe that these libraries generally do not have side effects that affect the typestate properties considered here, but clearly this omission is a potential source of unsoundness. Of course, when dealing with the full Java language, many other sources of unsoundness also apply, including concurrency, the potential for dynamic class loading of unanticipated code, and unmodelled native methods. Based on all these factors, we believe that excluding the GUI libraries is a reasonable choice for both a realistic tool and for this empirical evaluation. Fully sound precise verification of any property for the full Java language remains a difficult and unsolved problem.

The flow-sensitive combined typestate and alias analysis builds on a general Reps-Horwitz-Sagiv (RHS) IFDS tabulation solver implementation [Reps et al. 1995]. We have enhanced the standard IFDS solver in straightforward ways to handle Java's exceptional control-flow and polymorphic dispatch without undue precision loss.

6.2 Sparsification

To make the analysis scale, we rely on a lightweight sparsification[Ramalingam 2002] optimization prior to solving the IFDS problem. Consider an integrated verifier using access-paths bounded by depth k. We first consult the flow-insensitive points-to graph to conservatively determine all program variables that may appear in access-paths of depth at most k, which point to typestate objects of interest for a given property. Next, we perform a context-insensitive mod-ref analysis over the call graph, to determine those call graph nodes which may write to such variables; call these the *relevant* nodes. We prune the call graph to include *only* those nodes from which some *relevant* node is reachable, since the other nodes cannot modify the IFDS solution.

This pruning is particularly important for the LocalFocus verifier. Exploiting the pruning, the LocalFocus verifier can avoid making conservative assumptions for every method call, thus greatly increasing its precision.

We assume that methods from the standard libraries will not directly transition to *err*, and apply sparsification accordingly. This assumption introduces yet another

Table II. Call graph characteristics for benchmarks.

Benchmark	**Classes**	**Methods**	**Bytecode Stmts**	**Contexts**
bcel	751	4070	236,271	6011
gj	209	2253	131,288	2358
javacup	102	567	45,510	813
jbidwatcher	492	2723	180,492	3641
jlex	90	369	38,019	610
jpat-p	39	115	10,910	133
l2j	583	3443	209,184	4766
lucene	719	3540	224,478	5238
portecle	623	2992	210,543	4762
rhino-a	169	1150	81,388	1427
sablecc-j	362	2027	88,982	2476
schroeder-m	104	481	25,020	696
soot-c	651	2682	137,537	3105
specjvm98	627	3465	290,272	5654
symjpack-t	52	204	73,826	224
toba-s	132	610	52,985	838
tvla	331	1992	132,422	9331
total	**6036**	**32,683**	**2,169,127**	**52,083**

potential source of unsound results. We are not aware of any cases where this assumption does not hold for the typestate properties considered here. Of course, the analysis still must analyze all relevant library code to account for typestate transitions to non-*err* states, and aliases induced by the libraries.

In the staged verifier, we exploit results from early stages to improve sparsification in latter stages in two ways. First, if an early stage verifies that a particular statement does not transition to *err*, latter stages incorporate this information to improve sparsification. Second, if an early stage proves that a particular abstract object never causes an error, latter stages ignore tuples for that abstract object entirely.

6.3 Benchmarks

Table II lists the benchmarks employed in this study. Apache `Bcel` is a bytecode toolkit with a sample verifier. `Java_cup` and `JLex` are a parser generator and lexical analyzer, respectively, for Java. `Jbidwatcher` is an online auction tool. `L2j` is Multi-User Dungeon game server. Apache `lucence` is a text search engine. `Portecle` is a GUI application for managing secure keys and certificates. `SPECjvm98` is a collection of client-oriented applications. `TVLA` is a research vehicle for abstract interpretation. The remainder of the benchmarks come from the Ashes suite, described at the Ashes web page [6].

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. Recall that the context-sensitivity policy models some methods with multiple contexts.

[6]`http://www.sable.mcgill.ca/ashes/`

Table III. Typestate properties.

Name	Description
Enumeration	Call `hasNextElement` before `nextElement`
InputStream	Do not read from a *closed* `InputStream`
Iterator	Do not call `next` without first checking `hasNext`
KeyStore	Always initialize a `KeyStore` before using it
PrintStream	Do not use a *closed* `PrintStream`
PrintWriter	Do not use a *closed* `PrintWriter`
Signature	Follow initialization phases for `Signature`s
Socket	Do not use a `Socket` until it is *connected*
Stack	Do not `peek` or `pop` an empty `Stack`
URLConn	Illegal operation performed when already connected
Vector	Do not access elements of an empty `Vector`

Table III lists intuitive descriptions of the typestate properties verified in the experiments. For **Stack** and **Vector**, we check typestate automata to enforce that the program does not access elements of an empty collection directly after an explicit “clear” operation, or for newly created empty collections. Typestate cannot address the harder problem of verifying empty stacks or vectors in general.

6.4 Methodology

The experiments evaluate the following verification algorithms:

FI: flow-insensitive analysis (Sec. 2.2.1)

LocalFocus: the intraprocedural analysis (Sec. 2.2.2)

Base: the base analysis (Sec. 3.3)

Unique: the analysis using the *unique* reasoning (Sec. 4)

APMust: the integrated analysis without focus (Sec. 5)

APFocus: the integrated analysis with focus (Sec. 5.)

Staged: a staged analysis consisting of three stages: LocalFocus, Unique, and APFocus.

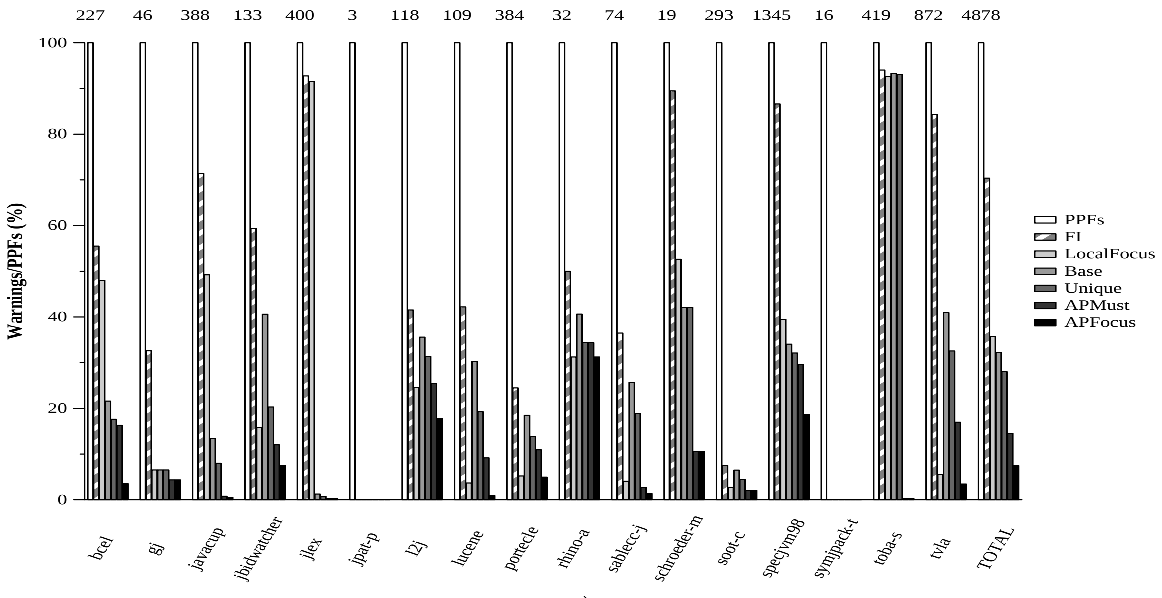
Note that each verifier performs the FI analysis as a first step, since it is extremely fast and can prune the workload based on the “verification scope” passed from the previous stage. The experiments use an access-path depth limit of 2, and unlimited access-path set width.

All experiments ran on an IBM Intellistation Z pro with two 3.06 GHz Intel Xeon CPUs and 3.62 GB of RAM, running Windows XP. The analysis implementation, consisting of roughly 200,000 lines of Java code, ran on the IBM J2RE 1.4.2 for Windows, with a max heap of 800MB.

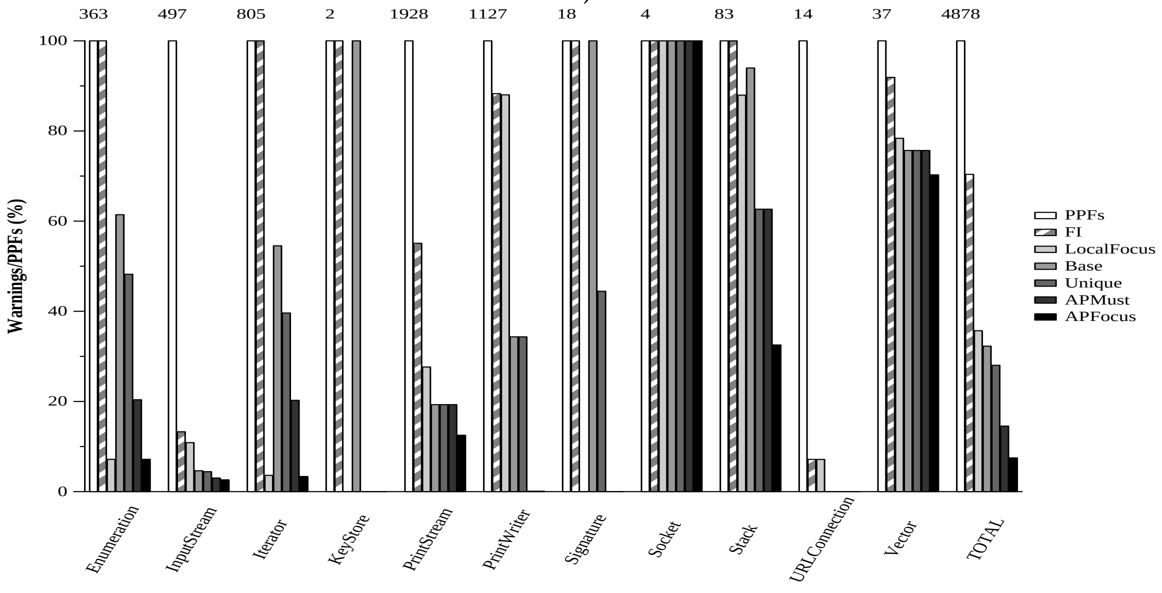
6.5 Results

Figure 6 shows the percentage of warnings, as a percentage of total number of statements that the callgraph indicates might transition to *err* (points of potential failure (PPF)). The number shown above each bar in the figure is the total number of PPFs.

The rightmost cluster of bars shows the total number of warnings across all runs. Overall,



a)



b)

Fig. 6. Percentage of warnings out of total number of points of potential failure (PPFs). Results are grouped by a) application, and b) property. Number of PPFs is shown above each group.

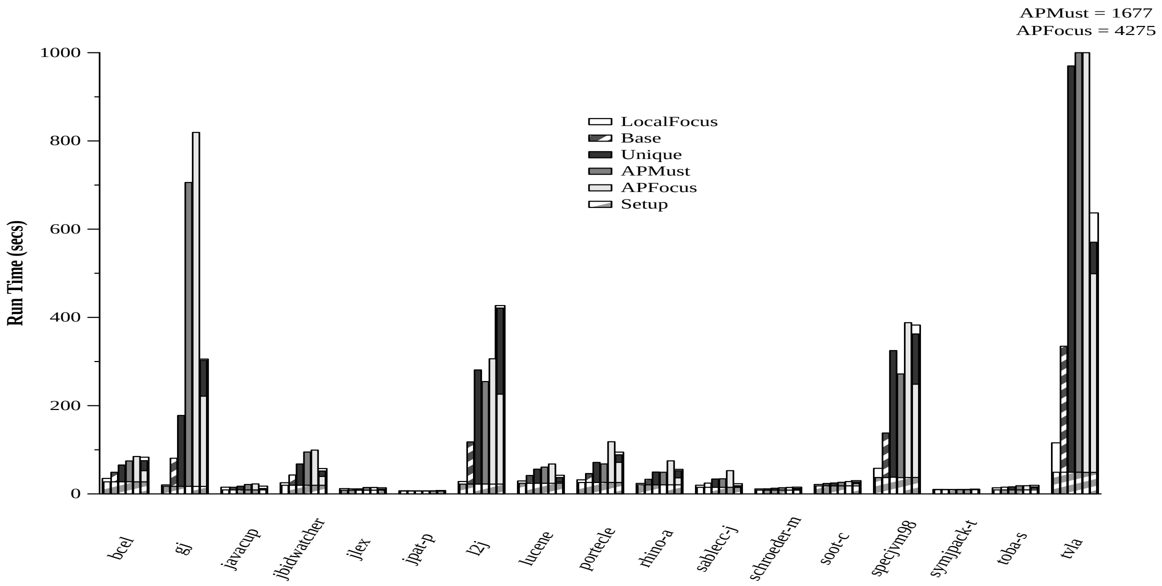


Fig. 7. Total wallclock time needed to run the analysis. "Setup" indicates the preliminary activities; primarily the preceding flow-insensitive pointer analysis and call graph construction. The rightmost stacked bar in each group represents the running time of the Staged verifier.

The FI verifier verifies correctness for **30%** of PPFs.

The LocalFocus verifier verifies correctness for **64%** of PPFs.

The Base verifier verifies correctness for **68%** of PPFs.

The Unique verifier verifies correctness for **72%** of PPFs.

The APMust verifier verifies correctness for **85%** of PPFs.

The APFocus verifier verifies correctness for **93%** of PPFs.

Table IV shows detailed results for verification warnings produced by the most precise (APFocus) solver. By construction, the Staged verifier has the same precision as APFocus. Sec. 6.7 discusses the sources of many false positives.

	Enum	InptStr	Itr	KStore	PrntStr	PrntWr	Sig	Socket	Stack	URLConn	Vector	Total	
bcel	0 / 2	0 / 1	0 / 15		0 / 36	0 / 139			8 / 32		0 / 2	**8 / 227**	**3.5%**
gj		2 / 6			0 / 40							**2 / 46**	**4.4%**
javacup	0 / 82	0 / 6			0 / 111	0 / 166			2 / 23			**2 / 388**	**0.5%**
jbidwatcher	1 / 8	0 / 9	0 / 46		0 / 31	0 / 9				0 / 13	9 / 17	**10 / 133**	**7.5%**
jlex	0 / 5				0 / 29	0 / 365			1 / 1			**1 / 400**	**0.3%**
jpat-p					0 / 3							**0 / 3**	**0.0%**
l2j		6 / 36	0 / 17		0 / 48			4 / 4	1 / 3		10 / 10	**21 / 118**	**17.8%**
lucene	0 / 29	0 / 6	0 / 11		0 / 60	0 / 1					1 / 2	**1 / 109**	**0.9%**
portecle	19 / 72	0 / 266	0 / 1	0 / 2	0 / 25		0 / 18					**19 / 384**	**5.0%**
rhino-a	3 / 9				0 / 16	1 / 1			6 / 6			**10 / 32**	**31.3%**
sablecc-j	0 / 24				0 / 47				1 / 3			**1 / 74**	**1.4%**
schroeder-m	0 / 6	2 / 11			0 / 2							**2 / 19**	**10.5%**
soot-c	0 / 14	0 / 2			0 / 213	0 / 58					6 / 6	**6 / 293**	**2.0%**
specjvm98	3 / 109	3 / 151			241 / 1075				4 / 9	0 / 1		**251 / 1345**	**18.7%**
symjpack-t					0 / 16							**0 / 16**	**0.0%**
toba-s	0 / 3	0 / 3			0 / 25	0 / 386			1 / 2			**1 / 419**	**0.2%**
tvla			27 / 715		0 / 151	0 / 2			3 / 4			**30 / 872**	**3.4%**
Total	**26 / 363** **7.2%**	**13 / 497** **2.6%**	**27 / 805** **3.4%**	**0 / 2** **0.0%**	**241 / 1928** **12.5%**	**1 / 1127** **0.1%**	**0 / 18** **0.0%**	**4 / 4** **100.0%**	**27 / 83** **32.5%**	**0 / 14** **0.0%**	**26 / 37** **70.3%**	**365 / 4878**	**7.5%**

Table IV. Findings for the most precise (staged) solver across all benchmarks and typestate properties. Each entry in the table shows the number of warnings as a fraction of the number of PPFs, for each benchmark/property combination.

6.6 Performance

Figure 7 reports the running times of the various verifiers across the benchmarks. The results show the expected relative costs of the various verifiers.

6.6.1 *Impact of Staging.* The Staged verifier improves performance compared to the APFocus verifier on 9 of the 10 codes where typestate checking takes more than 30 seconds. On these 10 codes, staging improves performance by up to 85% (`tvla`), with a median of 34%. Staging hurts performance by 40% on `l2j`; on this code, many PPFs survive early verification stages, and the cost/precision tradeoffs of the various solvers do not pay off.

6.6.2 *Impact of Sparsification.* We evaluated the sparsification of Sec. 6.2 across all runs of the staged verifier. With sparsification, 70% of supergraphs have fewer than 3500 nodes, 95% have fewer than 25,000 nodes, and 100% have fewer than 40,000 nodes. The corresponding numbers without sparsification are drastically higher: roughly 80% of unpruned supergraphs have more than 125,000 nodes, and 20% have over 290,000 nodes. Overall, sparsification reduces median supergraph size by roughly a factor of 50. We would expect a corresponding reduction in space and running time, if we could run the unpruned verifiers without running out of memory.

6.6.3 *Impact of Initial Pointer Analysis.* The precision of the preceding flow-insensitive pointer analysis significantly impacts performance and precision. A more accurate pointer analysis allows better sparsification, more effective live analysis and improved disambiguation overall. We ran many of the analyses with a context-*insensitive* Andersen-style pointer analysis, without the custom context-sensitivity policies described earlier. Many benchmarks timed out on several rules; we conclude that adequate precision in the preceding pointer analysis is vital.

Our context-sensitivity policy employs object-sensitivity for types from the standard libraries typically relevant to these typestate properties (namely collections and I/O streams). Some benchmarks defeat this object-sensitivity policy by using application-level collections or streams. For example, TVLA uses a library of application-level collections, and specJVM98 uses a reporting library of custom I/O streams. To handle these cases more effectively, we need to infer a pointer-analysis context-sensitivity policy for application classes that match typestate properties. Iterative refinement techniques [Plevyak and Chien 1994; Guyer and Lin 2003] may apply to this problem.

6.7 Discussion

Overall, the results show that our combination of techniques is relatively successful and efficient at verifying these typestate properties. The various techniques complement each other, contributing to the effectiveness of the staged verifier.

Since our goal in this paper is the successful verification of typestate properties, we have deliberately chosen a set of mature benchmarks. For our experiments, we assume that typestate violations in these benchmarks are all false alarms. We have examined, by hand, many of the warnings which our most precise verifier does not eliminate.

The *specJVM98* code's use of *PrintStream* accounts for 241 of the 365 warnings reported. These are all false positives, stemming from a few lines of code in the *specJVM98* harness. This program stores a *PrintStream* object in a static field *Context.out*, and uses the object ubiquitously throughout the various benchmarks. The particular idiom by which the program caches the *PrintStream* object in a static field defeats our focus heuristics, leading to a loss of precision.

Of the remaining 124 warnings, 53 arise from the *Vector* and *Stack* properties. Most of these warnings appear to represent a failure of the typestate property to capture all legal behavior, as opposed to solver limitations. For example, our typestate property for *Vector* does not account for the return value from *Vector.size()*. Many times, application code accesses a Vector via statements guarded by a test that $size > 0$. This pattern accounts for many of the false positives for the *Stack* and *Vector* rules. For proper treatment, these APIs require at least range-check analysis, as commonly applied to array-bounds checking (e.g. [Gupta 1993]).

The remaining warnings appear to arise from a combination of analysis approximations and typestate property limitations.

We expect that in the near future we can improve precision by a) access-path tracking for objects that are not typestate objects, but are likely to point to them, and b) increasing the scope of focus by exploiting inexpensive local alias reasoning. We suspect that substantial improvements in alias precision are within reach, without undue performance compromise.

In many cases, programmers deduce from application logic that a particular iterator must have a next element, or a particular collection must not be empty. The typestate property for a single object does not allow for application logic which ensures, via some back door, that an object occupies a particular typestate. Designing efficient, effective analysis for more general specifications remains a difficult problem.

7. RELATED WORK

Many existing verification frameworks (e.g., [Das et al. 2002; Ball et al. 2001; Corbett et al. 2000]) use a two-phased approach, performing points-to analysis as a preceding phase, followed by typestate checking. This approach only supports weak updates as discussed in Sec. 3.3.

The current version of ESP [Dor et al. 2004] uses an integrated approach, recording must and may alias information in a flow-sensitive manner. They observe that the may set becomes polluted and expensive to maintain, and even hint toward maintaining a must-not set as a possible future solution. In contrast, our approach adds must-not and also introduces the notions of uniqueness and focus, and uses staging to achieve increased scalability and precision.

DeLine and Fähndrich [DeLine and Fähndrich 2004] present a type system for typestate properties for objects. Their system guarantees that a program that typechecks has no typestate violations, and provides a modular, sound checker for object-oriented programs. To handle aliasing, they employ the *adoption* and *focus* operations to a linear type system, as described in [Fahndrich and DeLine 2002]. With these operations, the type checker can assume *must-alias* properties for a limited program scope, and thus apply strong updates allowing typestate transitions. Our approach can prove correctness of a more general class of programs, since a context-sensitive analysis can accept programs for which an expression cannot be assigned a unique type at a given program point. Furthermore, our *focus* operation generates facts that can flow across arbitrary program scopes, in contrast to the limited program scope handled by [Fahndrich and DeLine 2002]. On the other hand, our approach is non-modular and thus more expensive.

Degen et al. [Degen et al. 2007] present a type and effect system for a language called Java(X). Interestingly, this type system combines typestate type annotations with a notion of *droppability*, which force the programmer to retain pointers to objects in particular typestates, which could be used to protect against certain types of resource leaks.

Aiken et al. [Aiken et al. 2003] point out the importance of strong updates to avoid imprecision in the context of typestate verification (and other analyses). They present an inference algorithm for inferring *restricted* and *confined* pointers, which they use to enable strong updates. We believe that the *focusing* technique we exploit, inspired by [Sagiv et al. 2002], can sometimes achieve a similar effect without explicitly inferring *restricted* and *confined* pointers, and sometimes enable strong updates even when the pointers are not restricted/confined. Further, the uniqueness technique we use provides a somewhat orthogonal, cheap, technique for enabling strong updates.

Field et al. [Field et al. 2003] present algorithms based on abstractions that integrate alias and typestate information, but restricted to shallow programs, with only single-level pointers to typestate objects.

The parametric shape analysis presented in [Sagiv et al. 2002] has served as the basis for very precise verification algorithms, where the verification is integrated with heap analysis (e.g., [Yahav and Ramalingam 2004].) These algorithms, however, do not scale well. We plan to extend our staged verifier by adding such precise verifiers as a last stage.

Counter-example guided refinement [Ball and Rajamani 2002; Henzinger et al. 2002] based approaches have had impressive results in certain domains. But they have so far been less successful in dealing with complex heap manipulation, partly because these approaches attempt to *automatically* derive appropriate heap analyses. Our staged verifier has a "refinement" flavor, but restricted to a fixed set of manually crafted verifiers.

Aliasing of our combined domain resembles previous approaches to flow-sensitive, context-sensitive access-path-based pointer analysis [Landi and Ryder 1992; Choi et al. 1993]. Emami, Ghiya and Hendren [Emami et al. 1994] presented a domain that combined may and must points-to information. Our IFDS-based solvers memoize function summaries, similar to Wilson and Lam's partial transfer functions [Wilson and Lam 1995]. Our domain differs from these previous works since a) it tracks *must* and *must-not* paths, but not *may*, and b) Java's strong typing avoids complications arising from pointers to stack locations.

Iterative refinement techniques [Plevyak and Chien 1994; Guyer and Lin 2003] perform pointer analysis in multiple passes, with a client-independent first pass, followed by subsequent passes using context-sensitivity policies driven by client feedback. In future work we plan to integrate these techniques into our framework, where each typestate solver provides feedback for the next stage's pointer analysis.

Hackett and Rugina[Hackett and Rugina 2005] exploit a staged analysis to obtain a relatively scalable interprocedural shape analysis. This approach uses a scalable imprecise pointer-analysis to decompose the heap into a collection of independent locations. The abstraction used in [Hackett and Rugina 2005] is similar in spirit to our APMust abstraction, but the properties they target are more ambitious, namely proving the absence of memory errors.

Recently, [Shoham et al. 2007] presented a static analysis for client-side mining of temporal API specifications. Their approach is based on the heap abstractions described in this paper, but additionally use quotient-based abstractions for creating bounded descriptions of (unbounded) sequences of events.

Yorsh et al.[Yorsh et al. 2007] present a modular typestate analysis based on the abstractions introduced in this paper. Their approach is based on symbolic summarization and requires a decision procedure. We believe that a combination of our approaches has to be used in order to achieve a truly scalable typestate verification in the presence of aliasing.

8. CONCLUSIONS

We have presented a practical approach to typestate checking, exploiting staged analysis to realize precise alias analysis at reasonable cost. Results show that the techniques work well across a suite of programs, and indicate that efficient, precise typestate verification may prove valuable in the software development lifecycle.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for many helpful comments, and Mike Ernst for pointing out a bug in the original conference paper.

REFERENCES

Aiken, A., Foster, J. S., Kodumal, J., and Terauchi, T. 2003. Checking and inferring local non-aliasing. *ACM SIGPLAN Notices 38,* 5 (May), 129–140. In *Conference on Programming Language Design and Implementation (PLDI).*

Alur, R., Cerny, P., Madhusudan, P., and Nam, W. 2005. Synthesis of interface specifications for java classes. *SIGPLAN Not. 40,* 1, 98–109.

Andersen, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, Univ. of Copenhagen. (DIKU report 94/19).

Ashcraft, K. and Engler, D. 2002. Using programmer-written compiler extensions to catch security holes. In *Proc. IEEE Symp. on Security and Privacy.* Oakland, CA.

Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. 2001. Automatic predicate abstraction of C programs. In *Proc. ACM Conf. on Programming Language Design and Implementation.* 203–213.

Ball, T. and Rajamani, S. K. 2001. Automatically validating temporal safety properties of interfaces. In *SPIN 2001: SPIN Workshop.* LNCS 2057. 103–122.

Ball, T. and Rajamani, S. K. 2002. The SLAM project: debugging system software via static analysis. *ACM SIGPLAN Notices 37,* 1 (Jan.), 1–3.

Chase, D. R., Wegman, M., and Zadeck, F. K. 1990. Analysis of pointers and structures. *ACM SIGPLAN Notices 25,* 6 (June), 296–310. In *PLDI.*

Choi, J.-D., Burke, M., and Carini, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL 93.* 232–245.

Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., and Zheng, H. 2000. Bandera: Extracting finite-state models from Java source code. In *Proc. Intl. Conf. on Software Eng.* 439–448.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Proc. ACM Symp. on Principles of Programming Languages*. ACM Press, New York, NY, 269–282.

Crary, K., Walker, D., and Morrisett, G. 1999. Typed memory management in a calculus of capabilities. In *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM, Ed. ACM SIGPLAN Notices. ACM Press, New York, NY, USA, 262–275.

Das, M. 2000. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices 35,* 5 (May), 35–46. In *Conference on Programming Language Design and Implementation (PLDI).*

Das, M., Lerner, S., and Seigle, M. 2002. ESP: path-sensitive program verification in polynomial time. *ACM SIGPLAN Notices 37,* 5 (May), 57–68. In *Conference on Programming Language Design and Implementation (PLDI).*

Degen, M., Thiemann, P., and Wehr, S. 2007. Tracking linear and affine resources with Java(X). In *ECOOP 2007 Object-Oriented Programming*, E. Ernst, Ed. LNCS, vol. 4609. Springer-Verlag, 550–574.

DeLine, R. and Fähndrich, M. 2001. Enforcing high-level protocols in low-level software. In *Proc. ACM Conf. on Programming Language Design and Implementation*. 59–69.

DeLine, R. and Fähndrich, M. 2004. Typestates for objects. In *18th European Conference on Object-Oriented Programming (ECOOP)*. LNCS, vol. 3086.

Dor, N., Adams, S., Das, M., and Yang, Z. 2004. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*. 12–22.

Dwyer, M. B. and Clarke, L. A. 1994. Data flow analysis for verifying properties of concurrent programs. In *Proc. Second ACM SIGSOFT Symp. on Foundations of Software Engineering*. New Orleans, LA, 62–75.

Emami, M., Ghiya, R., and Hendren, L. J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices 29,* 6 (June), 242–256. In *Conference on Programming Language Design and Implementation (PLDI).*

Fahndrich, M. and DeLine, R. 2002. Adoption and focus: practical linear types for imperative programming. *ACM SIGPLAN Notices 37,* 5 (May), 13–24. In *Conference on Programming Language Design and Implementation (PLDI).*

Field, J., Goyal, D., Ramalingam, G., and Yahav, E. 2003. Typestate verification: Abstraction techniques and complexity results. In *Proc. of Static Analysis Symposium (SAS'03)*. LNCS, vol. 2694. Springer, 439–462.

Fink, S., Dolby, J., and Colby, L. 2004. Semi-automatic J2EE transaction configuration. Tech. Rep. RC23326, IBM.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. 2002. Extended static checking for java. In *Proc. ACM Conf. on Programming Language Design and Implementation*. Berlin, 234–245.

Foster, J. S., Terauchi, T., and Aiken, A. 2002. Flow-sensitive type qualifiers. In *Proc. ACM Conf. on Programming Language Design and Implementation*. Berlin, 1–12.

Grove, D. and Chambers, C. 2001. A framework for call graph construction algorithms. *Transactions on Programming Languages and Systems (TOPLAS) 23,* 6 (Nov.), 685–746.

Gupta, R. 1993. Optimizing array bound checks using flow analysis. *ACM Lett. Program. Lang. Syst. 2,* 1-4, 135–150.

Guyer, S. and Lin, C. 2003. Client-driven pointer analysis. In *Proc. of SAS'03*. LNCS, vol. 2694. 214–236.

Hackett, B. and Rugina, R. 2005. Region-based shape analysis with tracked locations. In *POPL*, J. Palsberg and M. Abadi, Eds. ACM, 310–323.

HASTI, R. AND HORWITZ, S. 1998. Using static single assignment form to improve flow-insensitive pointer analysis. *ACM SIGPLAN Notices 33,* 5 (May), 97–105. In *Conference on Programming Language Design and Implementation (PLDI).*

HEINTZE, N. AND TARDIEU, O. 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. *ACM SIGPLAN Notices 36,* 5 (May), 254–263. In *Conference on Programming Language Design and Implementation (PLDI).*

HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy abstraction. In *Symposium on Principles of Programming Languages.* 58–70.

LANDI, W. AND RYDER, B. G. 1992. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Notices 27,* 7 (July), 235–248. In *Conference on Programming Language Design and Implementation (PLDI).*

LHOTÁK, O. AND HENDREN, L. 2003. Scaling Java points-to analysis using SPARK. In *12th International Conference on Compiler Construction (CC).* LNCS, vol. 2622. 153–169.

LIVSHITS, B., WHALEY, J., AND LAM, M. S. 2005. Reflection analysis for java. In *Proceedings of Programming Languages and Systems: Third Asian Symposium, APLAS 2005.*

MILANOVA, A., ROUNTEV, A., AND RYDER, B. G. 2005. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol. 14,* 1, 1–41.

NAUMOVICH, G., AVRUNIN, G. S., AND CLARKE, L. A. 1999. Data flow analysis for checking properties of concurrent java programs. In *Proc. Intl. Conf. on Software Eng.* Los Angeles, 399–410.

PLEVYAK, J. AND CHIEN, A. A. 1994. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices 29,* 10 (Oct.), 324–324. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).*

RAMALINGAM, G. 2002. On sparse evaluation representations. *Theor. Comput. Sci. 277,* 1-2, 119–147.

RAMALINGAM, G., WARSHAVSKY, A., FIELD, J., GOYAL, D., AND SAGIV, M. 2002. Deriving specialized program analyses for certifying component-client conformance. In *Proc. ACM Conf. on Programming Language Design and Implementation.* ACM SIGPLAN Notices, vol. 37, 5. ACM Press, New York, 83–94.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proc. ACM Symp. on Principles of Programming Languages.* 49–61.

SAGIV, M., REPS, T., AND WILHELM, R. 2002. Parametric shape analysis via 3-valued logic. *Transactions on Programming Languages and Systems (TOPLAS) 24,* 3 (May), 217–298.

SHARIR, M. AND PNEULI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications.*

SHOHAM, S., YAHAV, E., FINK, S., AND PISTOIA, M. 2007. Static specification mining using automata-based abstractions. In *International Symposium on Software Testing and Analysis (ISSTA).*

STEENSGAARD, B. 1996. 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*, ACM, Ed. ACM Press, New York, NY, USA, 32–41. ACM order number: 549960.

STROM, R. E. AND YEMINI, S. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng. 12,* 1, 157–171.

WHALEY, J., MARTIN, M., AND LAM, M. 2002. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis.*

WILSON, R. P. AND LAM, M. S. 1995. Efficient context-sensitive pointer analysis for C programs. *ACM SIGPLAN Notices 30,* 6 (June), 1–12. In *Conference on Programming Language Design and Implementation (PLDI).*

YAHAV, E. AND RAMALINGAM, G. 2004. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation.* ACM Press, 25–34.

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

Received April 2007; no revision; accepted July 2007