RC24271 (W0705-140) May 30, 2007
Computer Science

IBM Research Report

Scaling Summary Computation of Large Java Libraries

Mangala Gowri Nanda
IBM India Research Laboratory

Satish Chandra
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Scaling Summary Computation of Large Java Libraries

Mangala Gowri Nanda
IBM India Research Laboratory
mgowri@in.ibm.com

Satish Chandra
IBM T. J. Watson Research Laboratory
satishchandra@us.ibm.com

ABSTRACT

We wish to pre-compute results of analyzing reachability of error conditions in large libraries. Our analysis uses exhaustive state-space exploration of an abstraction of the library code to discover all its possible behaviors, and represent them as method summaries. This exploration takes considerable time and space when analyzing large libraries. We present new techniques to speed up this exploration. Our techniques are based on empirical observations from trying to analyze large programs. Two of our techniques attempt to reduce the number of different environments under which a method must be explored, the first by identifying a dominant predicate that renders the other predicates immaterial to the outcome of a method, and the second by filtering out infeasible initial environments. A third technique re-uses the results of exploration of certain inner methods in a library by representing state information in a parameteric way. Together, these techniques enable us to compute reachability summaries of libraries with hundreds of methods.

1. INTRODUCTION

We wish to pre-compute, or *summarize* the reachability analysis of given error statements inside library code, in a modular fashion, without client code being available. The purpose of the analysis is to decide, conservatively, whether under some initial conditions it is possible for the execution to reach a given program point in library code. Answering this question in a non-trivial way requires a path-sensitive analysis—one that interprets conditionals depending on program state as opposed to non-deterministically—otherwise all program points in any reachable methods might appear to be reachable, modulo dead code. It also requires an abstraction powerful enough for reasoning precisely about heap references, because predicates on heap references could determine the reachability of an error statement. The novelty of our work is in doing summarization with this rich abstraction, *and* getting it to scale to real libraries.

.$

```
1  class File {
2    int state = CLOSED;
3    void open() {
4      if (state != CLOSED)
5        throw new Fail;
6      ...
7      state = OPEN;
8    }
9    void close() {
10     if (state != OPEN)
11       throw new Fail;
12     ...
13     state = CLOSED;
14   }
15   int read() {
16     if (state != OPEN)
17       throw new Fail;
18     ...
19   }
20 }
```

```
1  class ClientOfFile {
2    void main() {
3      File f =
4        new File("name.txt");
5      f.open();
6      f.close();
7      f.read(); // wrong
8    }
9  }
```

Figure 1: File class

Motivating Example Figure 1 presents the library code for a typical resource management protocol, and a small piece of client code. Suppose we are interested in the reachability of the error statement in `read()`. We consider a path-sensitive inter-procedural analysis starting from `main()`. At the call `f.read()`, this analysis would track that the read of `state` at line 16 is a read of `O.state`, where `O` is the object allocated in the client. The definition that reaches this program point is the definition at line 13, and the conditional expression at line 16 evaluates to true.

The reachability summary of the methods in `File` can be represented as a typestate specification for `File`, as shown in Figure 2. A typestate specification of a type is an automaton in which states correspond to some abstraction of the run-time state of an object of that type, and edges correspond to method invocations. For example, the automaton states in Figure 2 contain the predicates `(state == OPEN)` and `(state == CLOSED)` on a `File` object, which in this case happens to be the entire program-level state of the object.

Now consider the class `FileWrapper` shown in Figure 3. After line 5 of the client code shown, `fw.f` and `fl` are aliases. The state of the `File` object pointed to by `fw.f` after line 6 is *closed*, perhaps unintentionally so. The call `fw.read()` at line 7 will lead to an error. It is easy to see that the analysis here needs to be both path-sensitive as well as sensitive to aliasing conditions; furthermore, making the pessimistic assumption

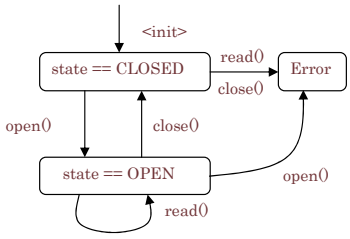

Figure 2: Typestates for File

in a summary that `fw.f` and `fl` are always aliases can be too conservative.

Richer Summaries In previous work [10], we introduced a generalization of single-object typestates to account for inter-object references. The idea is to encapsulate in an object's typestate, predicates not just on that object, but also on objects reachable by field dereferences (in an appropriately finitized way.) Furthermore, the transition diagram needs to account for aliasing behavior: different aliasing conditions cause different transitions to be taken out of a state, and furthermore, the transitions might generate aliasing conditions that a client verifier needs to keep track of. Figure 4 shows some of the transitions in the typestate specification for `FileWrapper`. Tracing through the code in `ClientOfFileWrapper` in this figure indeed leads to the error state.

Summary Creation We use systematic state-space exploration of the library to create summaries of this kind. This approach uses the reachability criteria provided by a user to discover relevant typestates—more precisely, the predicates that constitute typestates and that need to be handled relationally—and create a boolean abstraction [2] of the library code. Since a pre-pass of conservative pointer analysis is too imprecise for our purpose, we use a generalization of boolean abstraction, one in which heap references, including dynamic allocation, are modeled explicitly [10]. This generalized boolean abstraction requires a non-standard approach to state-space exploration, because the state space of the generalized boolean program is not bounded a priori. The summaries that we desire are the summaries of the methods in this generalized boolean program, obtained as a

```
1  class FileWrapper {
2    private File f = null;
3
4    void setFile(File g) {
5         if (f != null)
6            f.close();
7         f = g;
8    }
9    void done() {
10     if (f != null)
11        f.close();
12   }
13   int read() {
14     if (f != null)
15        f.read();
16   }
17 }
```

```
1  class ClientOfFileWrapper {
2    void main() {
3      FileWrapper fw = ...;
4      File fl = ...;
5      fw.setFile(fl);
6      fw.setFile(fl);
7      fw.read(); // wrong!
8    }
9  }
```

Figure 3: FileWrapper class

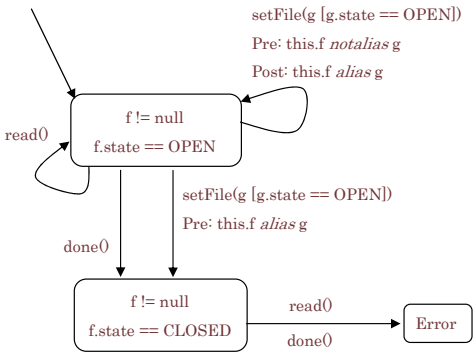

Figure 4: Some of the typestates of FileWrapper

result of the latter's state-space exploration.

Goal The goal of this paper is to make this summary computation scale to large Java packages that could contain hundreds of methods. A prototype implementation of the analysis presented in [10] works well for small libraries with a handful of classes, but runs into scalability problems very quickly because of combinatorial explosion in the state-space exploration phase. To address this problem, we consider techniques to improve the efficiency of the state-space exploration. Our ideal is to perform precise analysis for the given abstraction, so that no approximation is introduced in the exploration phase—the use of a summary in place of a whole-program analysis with the same abstraction should give the same answer to a reachability question.

Techniques for Optimizing Exploration Our techniques are based on empirical insights from large programs. Our first technique is to recognize that for any given method, not all predicates are equally important. It is often the case that valuations of certain "dominant" predicates entirely determine the outcome of a method in terms of whether it fails or not. If we can identify the dominant predicates, this can drastically reduce the exploration that needs to be carried out. Our second technique recognizes that predicate generation over a large library often gathers mutually-inconsistent predicates. When performing state-space exploration, simple satisfiability checks can be very effective in reducing the number of independent contexts that each method needs to be driven from. Our third technique is to leverage the fact that library methods often call other library methods in identical ways, and thus some computation can be cached and re-used during inter-procedural exploration. This is made possible by parameterizing the results of exploration of a method in a way that is independent of any particular calling context.

These techniques have enabled us in handling large Java packages such as java.io. As can be expected, there are scalability limits to relational analyses, and in some cases in these libraries, the multiplicity of predicates eventually forces us to coarsen the abstraction (the paper describes the particular approximation that we fall back to). The value of these techniques is in pushing the boundary of when that happens, so that larger libraries can be analyzed with as high a precision as possible. The design of good heuristics to coarsen the abstraction is outside the scope of the paper. Furthermore, prior knowledge of which classes in a library are not going to be directly used by a client can be helpful in regaining this lost precision.

Applications This work has several potential applications in software quality tools. (1) Precomputation of reachability analysis of library methods can save work in subsequent analysis of client code. (2) This technique can be used for static discovery, or "mining" of specifications of stateful objects. While scalable client code checkers have been constructed [6], a real bottleneck is non-availability of specifications to check. (3) The machinery for reasoning about preconditions for reachability of a particular program point can be used for creating targeted test drivers. While it is easy to create simple unit tests for methods, creating tests for large libraries is hard in terms of being able to drive the execution to specific program points.

Demonstration of these applications is outside the scope of the paper. However, we show that the summaries that we compute can be "consumed" in a client-code verifier. The analysis also helps identify interesting stateful classes in Java libraries, and their typestate specifications, although the precision of these specifications is limited by the precision of the abstraction we use.

Contributions The main contributions of this paper are as follows: (1) We present techniques for speeding up the computation of reachability summaries of the kind described above for large Java packages containing hundreds of methods. (2) We present an experimental evaluation of these techniques by performing summary computation on several large Java packages. (3) We also demonstrate the applicability of this approach in client-code verification. While our primary contribution has been in speeding up state-space exploration, just creating boolean abstractions of large Java library requires a significant engineering effort. In particular, we believe that flow- and context-sensitive pointer and escape analysis have not been reported previously on programs of this size.

We have *not* solved the problem of computing summaries for arbitrary static analysis problems; rather we have addressed this problem only for reachability analysis under a specific program abstraction. The strengths and limitations of this abstraction are discussed briefly in the paper. However, the focus on this paper is not on the power of the abstraction we use, but on improving the ability to apply it to larger libraries without losing precision relative to the specific program abstraction as far as possible.

Outline The rest of the paper is organized as follows. Section 2 discusses the most closely related work. Section 3 formalizes the problem of summary computation independently of client code. Section 4 describes the abstraction process as well as a base-line exploration algorithm. Section 5 describes techniques for scaling the state-space exploration. Section 6 describes our implementation and results on several large Java packages. Section 7 concludes the paper and discusses some future directions.

2. RELATED WORK

Sharir and Pnueli [16] describe a "functional" approach to interprocedural analysis in which for each function they compute a relation over input and output abstract states. Almost all techniques for summarization can be thought of as some realization of this general idea. The distinction between various techniques comes from the richness of the analysis domain, and from whether the summary is created by on-the-fly caching, or modularly in a bottom up fashion.

It is well-understood how to compute summaries for simple abstract domains that fit the IFDS criteria [11], for example as a graph reachability problem. This approach has been shown to scale to large programs. For pure boolean programs, scalable model checkers such as Bebop [3] have been used for creating summaries. Our analysis domain is different from either of these models, and so the above algorithms are not directly applicable. (Since everything is eventually finitized, one could fit this in a pure boolean model, but it would be needlessly large.)

Any path-sensitive analysis, modular or not, would need to identify a suitably rich domain, and this has been done in many ways, including heuristics [5], upfront predicate abstraction, like a single phase Blast [8] or SLAM [2], or iterative abstraction refinement, also used in Blast and SLAM. Identification of relational domain is an important problem in its own right. In a way, our idea of finding a dominant predicate already coarsens the abstraction in a local context.

Our work is also related to modular computation of summaries of points-to analysis. Much prior work has looked at computing modular pointer analysis; here we briefly mention three. Vivien et al. [17] have presented a compositional pointer and escape analysis, however they do not retain strong update information in their summaries. The work by Rountev et al. [14] generates summaries in the form of constraints. These summaries can take trigger iterative computation upon application. Our analysis goal is different, but in our dependence analysis, we also had to resort to assume a worst-case aliasing scenario, as it was not feasible to compute an exhaustive list of different input aliasing scenarios (though this does not effect eventual summary computation.) Chatterjee et al. [4] have presented a modular pointer analysis in which summaries are conditioned on the incoming aliasing context. Our technique borrows this idea from their work, but our analysis is richer since it is path sensitive, and also performs a richer analysis inside a procedure.

Our exploration strategy resembles the use of abstraction predicates used in the TVLA system [15], though we handle a far more restricted set of predicates compared to TVLA. Summarization has been studied in the context of TVLA as well [13]. Our goal is to be able to decide simpler properties, but make our analysis scale to large libraries.

As we indicated in the introduction, summaries are related to specifications of library code. With the recent success of static program verifiers [2, 6, 5], there has been a lot of interest in techniques to recover specifications from library code via static analysis [1, 19, 7]. The Jist [1] tool is perhaps the closest prior work related to our work. Jist uses a similar "tool chain" of predicate abstraction followed by exploration of boolean program; however their focus is on construction of bounded-sized and therefore more readable interfaces. In contrast, the focus of our work has been on efficiently constructing full interfaces for use mostly by tools, not humans. The work by Henzinger et al. [7] focuses on trying to find the right level of abstraction for the analysis of the library. It starts with a coarse abstraction, and then uses counterexample guided refinement to find a finer abstraction. We plan to explore the route of abstraction refinement in our future work as well. None of the static analysis-based previous works deals with inter-object references, and none deals with extracting specifications from Java packages in a fully automated way.

3. PROBLEM DEFINITION

We think of an application as consisting of the library code and the "client" code that uses the library. We wish to create summaries of the library code, so that the analysis of the client code can just make use of the pre-computed summaries.

The inputs to our analysis consists of the following: (1) C, a set of library classes. We assume that this set of classes is closed, in that it does not call any methods not in C. (This excludes analysis of certain framework-oriented code, where certain call-backs are bound later on.) (2) $A \subseteq C$, a set of API classes. These are the classes that would be instantiated in the client code. A could be same as C, but the analysis scales better if a narrower set of classes is identified as API classes. (3) E, a set of error statements whose reachability is to be tested for. E can be given generically in the form of all throw statements that raise a certain type of exception.

An execution—assumed to be sequential—of a Java program starts from `main()` method in a client T. The client interacts with the library in three ways: (1) It instantiates objects of classes in A via `new`. (2) It calls methods of classes in A, passing along arguments either of primitive types or of other classes in A, and (3) It gets return values from methods in A, which again, are either primitive types, or classes in A. Objects in A that are visible to the client code may have fields that refer to other objects in C, perhaps allocated internally inside the library. These objects are collectively called *library objects*. Note that the library objects do not contain pointers to objects not in C.

The *dynamic state* of library consists of the state of library objects and references between them. The execution of a method invocation depends on the initial library state, and determines the next library state. If an invocation reaches an statement in E, the post-invocation library state is a special *error* state. A *correct* execution of a client is such that it prevents the library from entering the error state. A correct client exhibits only correct executions. The job of a *static verifier* is to ensure that a client is correct, i.e. to flag any library method invocation in the client code that might possibly make the library enter its error state. A conservative approximation to this problem can be computed using a path-sensitive interprocedural analysis of the client T and the library C together.

Our problem statement is as follows: Given (C, A, E), compute a summary S that satisfies the following requirements: (1) For *any* client T, a static verifier could check T for correctness given just S, not A. (2) S should be constructed independent of any particular T.

One important issue here is the structure of S and the cost of using it in client-code verification. In the extreme, S could be just an intermediate representation of the code of C, e.g. in the form of constraints; however, then considerable work needs to be done in applying the effect of such a summary at a call site. We wish to extract summaries whose cost of application, ideally, is within a constant factor of the cost of applying the transfer function of a single Java statement during intra-procedural abstract interpretation.

4. BASELINE SUMMARY EXTRACTION

4.1 Overview of the analysis

Our analysis is driven by two key requirements: one, to determine reachability in a path-sensitive manner, and two, to perform analysis oblivious to any given client code (the client code may not even be available at the time of this analysis.) For the first requirement, we need to identify which predicates on instance variables of the library classes, or on method parameters, are the ones we want to keep track of in our static analysis. We do this using an upfront predicate abstraction using iterative weakest precondition computation. Due to the second requirement mentioned above, we need to simulate a synthetic client program that drives the (abstraction of) library classes through all possible behaviors in a finite way. This synthetic client program explores the state space of the library by running the library methods systematically on various heap configurations. The results of these explorations are recorded as summaries.

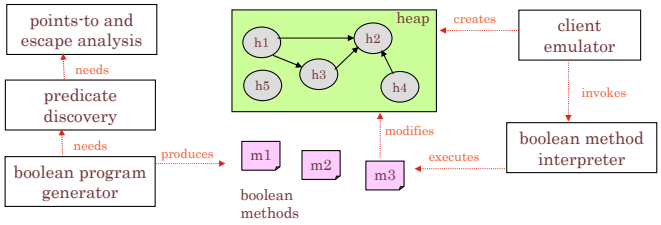

Figure 5: Overall process for summary computation

Figure 5 gives an overview of the approach. The various phases of this approach are described in the subsections that follow. Section 4.2 describes the left side of this figure, up to the generation of boolean methods; Since a detailed and formal description of this material is available in our previous paper [10], the treatment here is brief and informal. Section 4.3 describes the right side of the figure; the interpretation and client simulation described here are significantly different from our previous version. Section 4.4 desribes how to read off the summary from the client simulation.

4.2 Abstraction to Boolean Program

4.2.1 Dependence Analysis

The first phase of our analysis computes a combined pointer and escape analysis for the library in a modular way, starting bottom up in a call graph.[1] For each method, it computes a finite set of abstract heap locations to which each variable, local, parameter, or field, in the method refers. We name these abstract locations as follows: each reference parameter is assumed to point to a location F_i, which we call a *FormalIn*. Each reference field of a FormalIn points to an *EscapeIn* location, which we name as E_i; a reference field of an EscapeIn can point to an additional EscapeIn. Objects allocated at a `new` are named O_i, based on their allocation site in the code. We use 1-limiting to bound the number of locations that need to be created when traversing recursive structures. As a result, some E_i's and O_i's represent multiple concrete heap locations. Each variable is resolved to a set of abstract locations that it may refer to. This is done using a standard flow- and context-sensitive points-to analysis.

The dependence analysis of a method needs to know which FormalIn and EscapeIn locations may be aliased: rather than analyzing a combinatorially large number of different possibilities, we assume that all locations of the same type

[1]Analysis of mutually-recursive methods is done iteratively over the strongly connected component in the call graph.

may be aliased. As we shall see later, the precision of this dependence analysis does not cause a loss of precision in our eventual outcome; though it does matter to the efficiency of the analysis.

4.2.2 Predicate Discovery

The second phase discovers the predicates relevant for reachability analysis, and prepares for boolean method generation. We start from a particular label $l \in E$, which is typically a `throw`, to begin predicate discovery. The reachability of l depends on its immediately enclosing conditional, which is given by l's immediate ancestor in the control-dependence tree. We perform an iterative weakest pre-condition computation with respect to this seed conditional expression. We use standard rules for weakest pre-conditions, based on data dependence and substitution. Similarly to standard backwards slicing [9], we traverse both data and control-dependence edges when discovering more predicates of interest; we refer to this as *propagation*. The predicate generation is pretty simplistic, and at this time it does not interpret arithmetic expressions (for such statements, predicates involving the l.h.s. variable are assigned a non-deterministic value $*$).

Predicate generation is performed both intra-procedurally and inter-procedurally. For the latter, inter-procedural linkage computed in the earlier phase is used for propagation in the standard way. Interestingly, we also carry out propagation across methods that are not obviously related by a caller-callee relation, because dependence can flow through the (non-available) client space; we use type-based matching to posit such potential flows. For example, suppose a predicate $T.x > 0$ is read before being defined in a method m, and some other method n defines $T.x$, where T is some type. Then we have to propagate the predicate into n, because such flow might happen through the client. As we shall see later, this can add significantly to the state space that needs to be explored.

4.2.3 Boolean Methods

Once all relevant predicates have been identified, we abstract the library methods to boolean methods that manipulate these predicates instead of the original computation. However, unlike traditional predicate abstraction, we avoid hard-coding the results of the pointer analysis in the generated boolean code; the boolean code derived using a conservative pointer analysis would make the resultant summaries too conservative. Instead, we retain all reference copies in the boolean program. To take advantage of this generality, predicates in our boolean methods are parameterized over abstract locations. A predicate $L.f > 0$ is written as $P_n(t_i)$, where P_n is a 1-ary parametric predicate defined as $P_n(t : T) = (t.f > 0)$, and t_i is a variable that refers to location L. Predicates involving primitive variables in the original program do not take a parameter: this $x > 0$ is written as a 0-ary $P_m() = (x > 0)$.

Boolean methods contain: (1) reference parameters and boolean-valued parameters (corresponding to abstraction of primitive parameters), (2) reference assignments, putfields, getfields and allocation statements, (3) predicate assignments of 1-ary and 0-ary predicates in terms of other predicates or *, and (4) control flow of the original code, including invokes (calls) and returns.[2]

[2]As noted earlier, our full boolean program model also has 2-ary predicates, but we omit them for brevity.

```
1 setFile(f1: FileWrapper, f2: File, b1: boolean) {
2   if (P3(f1)) {
3         t = f1.f;
4         t.close();
5   }
6   f1.f = f2;
7   P3(f1) = b1;
8 }
```

`P3(fw: FileWrapper): fw.f != NULL`
Boolean `b1` represents the condition `f2 != null`

Figure 6: Part of the boolean code for File and FileWrapper

Figure 6 shows one of the boolean methods corresponding to `File` and `FileWrapper`. This example does not have an * assignment or 0-ary predicates as parameters, but an interested reader can find examples of these features in [10].

4.2.4 Limitations and Strengths of the Abstraction

At this point it is useful to consider the limitations and strengths of this abstraction. Among the major limitations are: (1) *Arithmetic* Since we do not model arithmetic, our predicates cannot track comparison between integer variables and constants (or between two variables). The outcome of any such conditional is always taken to be *. (2) *Arrays* Like most work on pointer analysis, we do not model different slots of an array separately. This limits our ability to say anything with respect to any data structure that uses an array internally. (3) *Recursive properties* Since our heap abstraction is 1-limited, we cannot reason precisely about properties of recursive data structures. In all these cases, the abstraction errs on the side of caution, that is to say, it will say some statement is reachable even if a more precise abstraction could have ruled out the reachability.

On the other hand, the abstraction is also quite powerful in dealing with equality predicates, e.g. $p.f == q$ on references precisely, as well as on other primitive predicates, e.g. $flag == 0$, that govern path sensitivity. It is worth reiterating that this abstraction does not depend on a fixed, conservative pointer analysis that we use for data dependence. An interpretation over our abstraction can compute a very precise path-sensitive alias analysis on the fly, because all heap operations are retained in the abstraction. As shall be seen in the next subsection, the advantage of this generalized boolean model is that the exploration can dynamically create as many predicate instances as needed to discover different behaviors, and only as many as needed (compared to exploring a heap of some worst case size determined a priori).

4.3 Interpretation and Client Simulation

In this phase we wish to drive the abstracted library classes through all possible behaviors. We do this in two parts: first, we describe how to compute all behaviors of a single method that starts from a given concrete state, and second, we describe a way to emulate a client that exercises all behaviors of the library.

4.3.1 Interpretation

The first part is an abstract interpretation problem, for

```
cs_init = initial concrete state based on parameter binding
WL = {(cs_init, startnode)}
AS(startnode) = {α(cs_init)}
AS(n ≠ startnode) = {}
while (WL ≠ {}) do
  pick and delete (cs, pp) from WL
  if (pp is exitnode) output cs and continue
  case stmt at pp of
    if (pred) pp-true pp-false:
      if cs implies pred
        add (cs, pp-true) to WL
      else
        add (cs, pp-false) to WL
    if (*) pp-true pp-false:
      add (cs, pp-true) to WL
      add (cs, pp-false) to WL
    throw e:
      output error
    default:
      cs-next = execute s on cs
      pp-next = next(stmt)
      AS(pp-next) = AS(pp-next) ∪ α(cs-next)
      if AS(pp-next) changed, add (cs-next, pp-next) to WL
end
```

Figure 7: Abstract Interpretation loop

which we need to design the abstract domain in a way that guarantees termination. The concrete execution state of a boolean method, at any program point in the method, consists of the following: (1) valuations of 0-ref predicates; (2) mapping of local reference variables to numbered heap objects, say H_1, H_2, etc. where each heap object also knows which of the abstract locations it belongs to, such as F_1, F_2, E_1 etc.; and, (3) valuation of 1-ref predicates on each of the heap objects, e.g. $P_1(H_1)$, $Q_1(H_1)$, $P_1(H_2)$, etc. Execution of a statement can impact valuation of predicates, and can also add a new H. Because a boolean method can create or traverse arbitrary heap data structures, its state space is not bounded, and some abstraction needs to be performed.

We define an abstraction α of a concrete execution state as follows. First, we define the *typestate* of an object as a evaluation of an ordered list of predicates, in which, first, there are the 1-ref predicates on that object, e.g. $P_1(this)$, $P_2(this)$, followed by 1-ref predicates (if any) on the object reachable by following each reference field of the object (if any), e.g. $P_3(this.f)$, $P_4(this.g)$, followed by 1-ref predicates on objects reachable by following two reference fields, e.g. $P_3(this.f.h)$, and so on. We prevent unbounded lists by ignoring objects reachable by paths in which field names repeat, e.g. $P_3(this.f.f)$ is not considered; our abstraction is not powerful enough to reason about shape properties. The abstraction creates a mapping from finitely many abstract locations to finitely many typestates: $AbstractLocation \mapsto Typestate$.[3] This mapping is the abstract execution state.

Figure 7 shows the iterative computation in abstract interpretation. It accumulates a monotonically increasing set AS of abstract execution states at each program point via iteration; since there are only finitely many abstract execution states possible, the iteration terminates. We employ a worklist WL in the iteration. Elements in WL are pairs of concrete states and program points. This is a useful optimization, because in our setting it is expensive to reconstitute concrete states from an abstract state, and statement transformers work only on concrete states. The method outputs concrete states reachable at the exit node. The method can also show a failing behavior from the given concrete initial state.

To handle procedure calls, we maintain a stack of worklists. When the interpreter enters a method, we push a new worklist on the stack with the initial pair (cs, startnode_p) in it, where cs is the state at the point of the call, including the parameter bindings. Upon return, we restore WL from the stack, adding (cs, pp-next), where cs is the state at the end at return, excluding the parameter bindings, and pp-next is the statement following the call statement. This basic scheme is unable to handle recursion. In future, we plan to explore techniques such as the one proposed by Rinetzky [12] to support recursion.

4.3.2 Client Emulation

The second part of this phase concerns client emulation. The purpose of the client emulator is to execute each method in all distinct initial states of interest, and produce the output states. The emulator works as follows for each API method $m \in A$ (steps 2, 3, 4 have to be repeated for different choices):

1. Let m have reference parameters $p_1, \ldots, p_n$ and boolean parameters $b_1, \ldots, b_m$
2. Choose a typestate for each p_i
3. Choose true or false value for each b_i
4. Choose an aliasing among abstract locations (F_i, E_i etc) of m
5. Instantiate m's formal parameters with a concrete state conforming to choice made in steps 3, 4, and 5
6. Run the abstract interpretation loop on m (see Figure 7)

Not all typestates are necessarily reachable, so we iterate over only the ones that are found so far in the output states.

We believe that this exploration procedure is a substantial improvement over the one used in our previous work. In that work, our idea was to explore a certain fixed-sized configuration of heap objects fully, and then grow this configuration to find more behaviors. It was tricky to argue when we had explored all behaviors, and moreover, the implementation of the abstract interpretation loop also had special cases because `new()` could not always return an fresh object because of the upfront heap limitation. In the present scheme, the finitization comes from the termination of the abstraction we apply in the interpretation loop, rather than by upfront finitization of heap.

4.3.3 Resorting to Approximations

Despite the optimizations we present in Section 5, the exploration may run out of resources for large enough libraries. In such a situation, we need to resort to approximations (this approximation is relative to a fully relational analysis.) When we use approximations, the summaries that we compute could be less precise than the whole-program analysis under the same abstraction.

One way to approximate is to reduce the number of distinct elements possible in AS. For this approximation, we alter the definition of typestate computation to account for predicates reachable in k steps. Since fewer distinct typestates are possible, this is a coarsening of the abstraction.

Sometimes the number of initial bindings that the em-

[3] As a matter of detail, the abstraction maps locations that represent multiple concrete heap objects to a set of typestates.

ulator must try for a certain method becomes too large, typically because of a large number of typestates in more than one reference parameter (even at $k = 0$). In this case, we resort to approximating the results by binding a special * typestate (meaning all constituent predicates are set to *) for some of the formal parameters and * for boolean parameters.

4.4 Extracting the Summary

Each run of the interpretation of a method is a transformation of an input state to possibly many output states. For each output state produced by the interpretation, we read off typestate changes of all the abstract locations of that method, and we read off the pertinent reference field changes from the output. Thus the format of a transition for a method m is:[4]

$$(AbstractLocation \mapsto Typestate, Aliasing) \xrightarrow{m} (AbstractLocation \mapsto Typestate, Aliasing)$$

Here *Aliasing* is a partition on *AbstractLocation*, e.g., if E_1 is defined as $F_1.f$, and F_2 and $F_1.f$ are aliases, then F_2 and E_1 are in the same partition. The partition in the post-condition of the method can obviously be different from the one in the pre-condition. The set of all transitions of a method is its summary.

5. OPTIMIZING THE EXPLORATION

We now describe three techniques for speeding up the exploration phase of our analysis. These techniques do not introduce any approximation by themselves.

5.1 Hierarchical Exploration

Our base-line client emulation attempts to bind each reference parameter with each of the known typestates of the respective type. This can, and often does lead to too many combinations. We leverage the empirical observation that in many methods there is a "dominant" conditional, which when true, always causes a method to fail, irrespective of the valuations of other predicates of that type. If so, then there is little point in exploring the method with respect to all possible different parameter bindings.

```
1 m(p1,...,pn,b1,...,bm) {
2   ...
3   if (P(pk))
4     throw e
5   ...
6 }
```

Depending on the valuation of $P(p_k)$, the other predicates on p_k do not matter, and neither do the typestates of the other p_i's or values of b_i's, the boolean parameters. Therefore, those typestate of p_k that have the predicate P set to true all lead to failure, regardless of other parameters of the method, and in that situation, the emulator not need to try exhaustive bindings of parameters of m.

The new client emulation works as follows. First we try to identify such dominant predicates (steps 2 and 3 have to be repeated for different choices):

1. Let m have reference parameters $p_1, \ldots, p_n$ and boolean parameters $b_1, \ldots, b_m$
2. Choose a p_k as the object
3. Choose a typestate for p_k
4. Assign * typestate for each p_i, where $i \neq k$, and assign * for each b_i
5. Choose an aliasing among abstract locations (F_i, E_i etc) of m
6. Instantiate m's formal parameters with a concrete state conforming to choice made in steps 2, 3, and 5
7. Run the abstract interpretation loop on m (see Figure 7)
8. If all outcomes for p_k with the typestate selected in step 3 are failing, make that typestate for p_k as 'dominant'

Next we run the original client emulation loop, except at step 2 we skip the typestates already found to be dominant. The additional insight used here is that not all predicates of formal parameters appear in conditional expressions in a method, so it is not the case that setting some of those predicates to a * value would only merely defer trying out both true and false cases during interpretation of the method.

One may ask why not detect these situations statically, using dependence analysis. This is indeed possible in some simple cases, but is harder to implement, compared to leveraging the interpreter that we need anyway, and is less general compared to the "dynamic" analysis in the interpreter.

5.2 Feasibility Checks on Typestates

This optimization has to do with the way in which predicates comprising a typestate are generated.

Recall from Section 4.2 that the predicate discovery phase does a backwards traversal of the forest of call graphs and generates predicates for each class. Since some classes are instantiated in many different contexts, they club unrelated predicates into the same class.[5] For example, suppose the class `String` has a integer variable `count`, and that code that instantiates a `String` can get at its `count` via a get method. This value could be used in a comparison—a predicate of the form *count relop c*—at a number of unrelated contexts. All of these predicates get associated with the class `String`. It turns out that many of these predicates are mutually inconsistent: e.g. $count < 5$ and $count > 32$.

At other times, since our predicate generation assumes a possible flow of value via a unknown client based on type-based matching, predicates get added to the class that would not have otherwise been added had the whole program been known; clearly these can give rise to unrelated predicates in a single class, in the same manner as above.

Sometimes such redundancy could even appear inside a single class: e.g. `state == OPEN` and `state == CLOSED` cannot simultaneously be true if `OPEN` and `CLOSED` are unequal literals.[6]

Effectively, there are a far fewer typestates of some types that are actually possible, than a simple-minded independent assignment of values to predicates would suggest. The modified client emulator takes advantage of this observation as follows: At the step in which it initializes each of its reference parameters to one of the known typestates of its type, it considers only those typestates in which the constituent predicates are mutually consistent. At this time, the fea-

[4]Again, as a matter of detail, some abstract locations may represent multiple concrete objects; in that case, that location maps to an (unordered) set of typestates.

[5]This situation is somewhat reminiscent the problem that object sensitivity in pointer analysis addresses.

[6]We cannot say the same for `state != OPEN` and `state != CLOSED` unless we know `state` ranges over exactly those two values.

sibility checks are implemented on the following predicate families: integer comparisons, string comparions, and instanceOf predicates.

While this was simple and extremely profitable optimization, we did not realize its need until we examined the boolean programs by hand to see why the client emulator was running through so many different initial conditions for some of the methods.

5.3 Parametric Exploration

Recall that at procedure calls in the exploration of boolean methods, we simply stack the worklists and continue exploration in the called method. It is desirable to be able to re-use the results of exploration of a called method, if the latter has been analyzed previously with similar initial conditions. The main problem in achieving this re-use in the base-line exploration algorithm is that our interpretation in terms of concrete states; the abstract execution states were used primarily as a device for ensuring termination during the interpretation loop.

However, realize that the behavior of a method is still "parametric" in which exact set of heap objects it works on, modulo renaming of those objects.

Given the abstract locations F_i's and E_i's of a method m, and given an aliasing partition on these abstract locations at an invocation consisting of n equivalence classes, we come up with a labeling of distinct objects to *positions* numbered $1..n$. Suppose a method is invoked with $H_1, \ldots, H_n$, where H_i is the concrete object at position i, and suppose there is another invocation with $H'_1, \ldots, H'_n$ where H'_i is the concrete object at position i. If it is the case that for each $i = 1..n$, for each 1-ary predicate P defined on the type of H_i, $P(H_i) = P(H'_i)$, then, the effect of invoking the method on H_i's and H'_i's would be identical, modulo renaming. Therefore, the behavior of a method invocation can be expressed parametrically in terms of positions: the effect of invocation is in updating the predicates of these objects to new values, and fields to different objects, which are referred to by their original position.

Since objects can be allocated inside the method, the effect of an invocation needs to mention these objects as well; these objects are assigned positions corresponding to locations O_i's, using numbers $n + 1$ onwards. Note that the position of an object is a constant for the duration of a method.

The parametric behavior of a method, then, is of the form, where the assignment of position implicitly accounts for fields in the pre-condition.

$$Position \mapsto \{Predicates \mapsto \{\top, \bot\}\} \longrightarrow$$
$$Position \mapsto \{Predicates \mapsto \{\top, \bot\}, fields \mapsto Position\}$$

During exploration, we maintain a table of such input to output mappings. When we arrive at an invocation, we check if the pre-conditions of some entry in the table matches—if so, we simply need to update the predicates and fields of the corresponding concrete objects as dictated by the post-condition in that entry. If not, we perform the interpretation of the method and insert a new entry, or entries, in the table. Maintaining such a table is the standard function points approach to inter-procedural analysis [16].

At this point, it is tempting to think that we can perform our entire analysis in a layered manner, first generating summaries of the leaf classes in the call graph of a library, and the using these summaries in the caller of those leaf classes. The catch, though, is that the predicate discovery phase might lead to additional predicates in the called classes begin discovered during the analysis of caller classes. We need to perform a "whole library" predicate discovery!

6. RESULTS

We have implemented our analysis using the WALA [18] Java byte-code analysis infrastructure. Our implementation on top of WALA consists of approximately 35,000 lines of Java code.

We list some of the assumptions made in the implementation. We do not deal with control-flow through exceptions, and instead consider `throw` statements as error statements. Since library code might refer to some abstract classes, we assume a certain concrete class for each case: assuming all possible concrete classes everywhere renders the analysis useless (though in practice one can customize the analysis with respect to which concrete classes are provided.) The analysis cannot deal with the case where no concrete class corresponding to an abstract class exists in the library. For the exploration phase, at some places the chain of dynamic calls is so deep that we truncate methods that are beyond a certain dynamic depth (> 7) from any API method (an alternative would be to create a list of "ignore" methods where the analysis bottoms out.)

6.1 Scaling of Summary Computation

In this section we briefly describe the characteristics of some Java libraries that we tested our analysis on. We then give the results of the analysis, though due to space limitations we do not show the transition diagrams. We ran the examples on an Intel based Linux machine (2.2 GHz, 2GB RAM). We report the time to execute the analysis.

Sample APIs. We ran our analysis on java.io, which is a standard, and fairly large library. We also ran our analysis on four other comparatively small libraries, java.util.jar, java.util.zip, java.awt.color and java.awt.font. The characteristics of the APIs are given in Table 1. The column labeled "totC" gives the total number of classes analyzed. This includes classes from other libraries that are required for the analysis. For example, java.util.jar uses many classes from java.io. The column labeled "totM" gives the total number of methods in the call graph; "nThrow" gives the number of throw statements in the call graph which form the basis for computing the slice; "slicedM" gives the number of methods in the slice; "apiM" gives the number of public methods in the API that are in the slice. These are the methods that are accessible from a client program. The column labeled "nS" gives the total number of bytecodes that have gone in to build the call graph. The column labeld "tPrelim" gives the time in seconds to generate the boolean methods.

Scalability Analysis. We explored three techniques for improving the scalability and performance - hierarchical analysis, feasibility checking and parametric exploration. In Table 2, the column labeled "Opt" has three possible entries - "H" for hierarchical analysis, "H+F" for hierarchical analysis along with feasibility checks and "H+F+P" for the combination of hierarchical, feasibility and parametric ex-

Benchmark	totC	totM	nThrow	slicedM	apiM	nStmts	tPrelim
java.io	163	2888	694	1887	851	516178	8.9
util.jar	26	1428	300	824	136	468913	1.2
util.zip	33	535	167	388	167	17663	0.5
awt.color	23	335	87	188	75	11359	0.7
awt.font	51	1512	221	911	268	55774	9.1

Table 1: API Characteristics

Opt	time	#trans	Mtrans	avePre	maxPre	#TS	maxTS
java.io							
H+F+P	2949	22569	12	1.8	25	2957	270
util.utiljar							
H	93	3822	2	3.8	22	1144	325
H+F	339	1084	5	5.0	22	266	75
H+F+P	52	1084	5	5.0	22	266	75
util.utilzip							
H	39	3745	3	1.3	9	814	748
H+F	68	10357	3	1.3	9	650	534
H+F+P	30	10357	3	1.3	9	650	534
awt.color							
H	12	1832	4	6.9	16	69	13
H+F	21	2049	3	7.1	16	63	11
H+F+P	29	2049	3	7.1	16	63	11
awt.font							
H	1782	6099	3	18.2	56	1036	258
H+F	978	15283	3	8.3	56	1426	420
H+F+P	721	15283	3	8.3	56	1426	420

Table 2: Sizes of summaries

ploration. The column "time" gives the time in seconds. The column "#trans" gives the total number of transitions and "Mtrans" gives the maximum number of transitions in any one given method. The column "avePre" gives the average number of preconditions (boolean comparisons) for each transition and "maxPre" gives the maximum number of preconditions encountered in any transition. The significance of the size of transitions is in the cost of selecting the transition to apply when in a client verification application (see next subsection). The column "#TS" gives the total number of typestates computed across all classes and "maxTS" gives the maximum number of typestates for any one class.

For java.io, the computation did not complete in a reasonable amount of time except with all the optimizations (it also needed a lowered threshold of 256 of maximum number of initial environments). As can be seen from the table, with feasibility checking, the number of typestates and transitions becomes much smaller as redundant states are not generated. The time to analyze, however increases due to the overheads of feasibility checking in the smaller examples; though the time to analyze significantly reduces in java.font, as the computation overheads are compensated by the smaller number of typestates and transitions. With parametric exploration, there is no improvement in the precision, though there is upto 80% reduction in time to analyze. The performance benefit of parametric exploration is an indirect evidence that a client verifier would improve performance by using summaries of library methods.

In order to control the number of combinations generated to seed a method, we perform hierarchical analysis. Table 3 gives the result of hierarchical analysis. The column "#Hier" gives the number of times the hierarchical optimization was applied (when the number of initial combinations was over a threshold of 512), "max#Combs(H)" gives the worst case for the number of initial combinations. Often the analysis succeeds in bringing the number of combinations to within a chosen threshold (512 in our experiments). Column "post#Combs(H)" gives the number of combinations when the hierarchical analysis succeeds. Note that hierarchical analysis does not lead to any imprecision.

However, sometimes the hierarchical analysis does not reduce the number of combinations sufficiently and we are forced to apply approximation, where some parameter types are set to "*". In Table 3, column "#approx" gives the number of times the analysis needed to resort to this approximation. Column "max#Combs"is the worst case result of hierarchical analysis which is the input to the approximation analysis. Column "post#Combs" gives the number of reduced combinations after the approximation. There is a loss of precision incurred in this process. However, we observed that the analysis resorted to this approximation less than 3.5% of the time for java.io, and 0.5% of the time for others.

As can be seen from the table, it is possible for the number of combinations to grow exponentially. To understand the source of this explosion, notice that the maximum number of typestates a type can have was as high as 270 (in java.io), although an overwhelming majority of types had only 2 typestates. It takes a combination of only two or three objects with many typestates and a string of predicates takes the number of combinations close to the size of a long integer!

Benchmark	#Hier	max#Comb(H)	post#Combs(H)	#approx	max#Combs	post#Combs
java.io	1771	512454202490880	252	693	1719744	52
util.jar	76	622592	90	3	622592	512
util.zip	87	2604	347	21	2232	6
awt.color	31	23191344	324	7	1756920	121
awt.font	578	1924327342080	250	58	63252	252

Table 3: Scaling

To get some idea about the quality of results, we also measured the average number of post-conditions per pre-condition of a method. We found that an overwhelming number of times (close to 90%), there is a precise case of exactly one post-condition for a given pre-condition. The maximum number of post conditions for a given pre-condition is not larger than 12 for java.io, despite having to perform conservative analysis for some methods. The java.awt and java.util.jar have a maximum fanout of 5 and 4 respectively.

Localized Analysis. Our tool identifies the number of typestates for an object and hence can be used to *mine* for objects that show stateful behavior and then analyze them in isolation. Localized analysis optimistically assumes that the client program does not instantiate other types of the API in question, though the analyzed type may of course call methods of other types. This can be very useful as a program understanding tool. In Table 4, we give the results of localized analysis for a few of the stateful objects we discovered.

Since this is an exercise in program understanding rather than generation of complete transition information, it is possible to start with a subset of the throw statements in the slice. This shows the transitions that lead to a particular throw (or subset of throw statements) and hence is very useful; for example, for PipedReader and PipedWriter classes we experimented with four different set of throw statements that generated four different set of transitions.

In Table 4, we show the set of transitions that were generated by using all throw statements in the respective example as the seed. The table gives the number of API classes analyzed ("apiCl"), the number of API methods analyzed ("apiMeth"), the time (in seconds) for the analysis, the number of transitions ("#trans") and the number of TypeStates discovered ("#TS").

The smaller example (java.io.Piped* and java.io.Buffer*) outputs are actually human-readable. However, due to lack of space we do not give the output.

6.2 Using Summaries in a Client Verifier

We describe an extension of our tool for client-code verification that makes use of the summaries created as described earlier. The use of these summaries require co-operation from the client-code verifier, and we did not have a suitable existing verifier in which to add summary usage. Instead, we leverage our boolean program interpreter to serve the purpose of a client-code verifier as well.

First, we make our summary computation output transitions of each method as a synthetic Java method that encodes the transitions in a stylized way; this is shown by the example of `setFile`.[7]

```
1  setFile(FileXX t2) {
2    FileWrapperXX t1 = this;
3    ...
4    else if((t1.__TS__==2) && (t2 == null)){
5     t1.__TS__=0;
6     t1.f.__TS__=1;
7     t1.f=null;
8     t2=null;
9    }
10   else if((t1.__TS__==2) && (t2.__TS__==0)
11           && (t1.f==t2)){
12    t1.__TS__=1;
13    t1.f.__TS__=1;
14    t1.f.__TS__=1;
15   }
16   else if((t1.__TS__==2) && (t2.__TS__==0)){
17    t1.f.__TS__=1;
18    t1.f=t2;
19   }
20   ...
21 }
```

Note that the type definitions `FileXX` and `FileWrapperXX` need to be changed to track a typestate field.

The client code has to be modified in three ways: (1) it has to instantiate the modified types instead of the real types, (2) it has to invoke synthesized summary methods rather than actual library method, and (3) it needs to apply a type-specific "update" function to refresh typestates after a method call on a reachable object. For example, if the client modifies a `FileXX` object, it needs to call an typestate update function on each `FileWrapperXX` object so that the latter's typestate is current. The update code for `FileWrapperXX` is as shown here:

```
1 void __update__() {
2   ...
3   if(((f == null)==false) && ((f.__TS__==1) ) ){
4    this.__TS__ = 1;
5   }
6   ...
7 }
```

We now apply the same tool chain of abstraction and interpretation to this program to see if `main()` of the client code has an error transition. There is no need for trying different bindings of typestate for parameters, because we only need to drive the execution from a `main()` method. The interpreter computes fix point in the same way as when creating summaries, except it only needs to find an error transition, and not output all transitions.

The format of these synthetic methods that describe transitions was designed so that the predicate abstraction phase can recover the transitions losslessly. In other words, if we create a summary of the above method, we would get back the same method!

This verifier has been tested on small examples only. It is

[7]It is clear that this code can be smaller in size and more efficient if generated in a nested style. This is being implemented; the code shown here is as currently generated.

Example	#apiCl	#apiMeth	time	#trans	#TS
java/io/Buffer*	3	62	2.0	178	16
java/io/Piped*	4	34	2.3	656	51
java/io/*File*	10	255	119	4845	376

Table 4: Localized Analysis

designed only as a proof of consumability of our summaries. In practice, a scalable client-code verifier would use much more efficient abstractions in the client space and do just enough book-keeping to use summaries.

7. CONCLUSION AND FUTURE WORK

The goal of this paper was to improve the scalability of the state-space exploration, during summary creation of large libraries. We presented three techniques towards this end. The first two techniques reduced the number of different starting configurations from which a method's state space needed to be explored. The third technique used a parametric representation of state to achieve reuse of computation within summary creating. Together these technique helped us to create summaries for libraries with hundreds of methods, as shown by our experimental results. We also showed how to use our boolean program interpreter in a novel way to create a client-code verifier that could use the kind of summaries we create.

It is clear that for large libraries we need to resort to approximations. We plan to explore better techniques for predicate generation to try not to get too many predicates in the typestates of library objects. We also plan to investigate heuristics that work better than the simple-minded approximations that we fall back on at the moment. Finally, we wish to explore using our summaries with a practical client-code verifier.

8. REFERENCES

[1] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 98–109. ACM Press, 2005.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103+, 2001.

[3] T. Ball and S. K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *PASTE*, pages 97–103, 2001.

[4] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *POPL*, pages 133–146, New York, NY, USA, 1999. ACM Press.

[5] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002.*, 2002.

[6] S. Fink, E. Yahav, G. Ramalingam, N. Dor, and E. Gaey. Effective typestate verification in the presence of aliasing. In *ISSTA*, 2006.

[7] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proceedings of ACM Conference Foundations of Software Engineering*, 2005.

[8] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *In Tenth International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, 2003.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *Proceedings of the Sigplan '88 Conference on Programming Language Design and Implementation*, 23(7):35–46, July 1988. Atlanta, Georgia.

[10] M. G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *OOPSLA*, pages 77–96, 2005.

[11] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[12] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. *Lecture Notes in Computer Science*, 2027:133–149, 2001.

[13] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, pages 284–302, 2005.

[14] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. *Lecture Notes in Computer Science*, 2027:20+, 2001.

[15] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[16] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S.Muchnick and N.D.Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[17] F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2001.

[18] T. J. Watson Libraries for Analysis. http://wala.sourceforge.net.

[19] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium of Software Testing and Analysis.*, 2002.