Deriving Object Typestates in the Presence of Inter-Object References

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

Christian Grothoff
University of California Los Angeles
christian@grothoff.org

Satish Chandra
IBM India Research Laboratory
satishchandra@in.ibm.com

ABSTRACT

We are interested in static analysis of Java classes with the goal of discovering the preconditions under which a certain program point within a method may be reached, taking into account the effects of previous method calls on an object of that class. The information pertinent to this computation is represented as the object's typestate, which is a finite set of relevant predicates that abstract the object's actual state. The execution of a method depends on an object's current typestate as well as other input parameters; the object may transition to a different typestate during the method's execution.

It is common for objects to contain references to other objects. In such cases, an object's behavior may depend on, in addition to its own state, the state of objects it has a reference to. The main contribution of this paper is to discover relevant object typestates, as well as transitions between typestates, in the presence of inter-object references. Our analysis first performs a combined predicate discovery and predicate abstraction to derive "boolean" versions of Java classes given as input. It then uses abstract interpretation to compute the typestate transitions caused by method calls. A novel aspect of this work is that a set of Java classes is analyzed in isolation, without any client program being provided. To do this, the analysis simulates all possible client's actions via a synthetic heap, all of whose interesting configurations are explored by our analysis.

The information we compute can be put to use in several ways. It can be used in checking whether a given client code erroneously uses a set of Java classes in a way that can throw an exception. It can also be used in creating test drivers for Java classes in order to exercise all relevant code paths in the corresponding methods.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]:

OOPSLA '05, October 16–20, 2005, San Diego, California, USA.

General Terms

Languages, Verification

Keywords

Java, alias analysis, interface specification, predicate abstraction, heap analysis

1. INTRODUCTION

It is common to develop and distribute software components as building blocks of large software systems. For the purpose of this paper, a software component is a set of Java classes that are closed in the sense that the target of any call within the component is also part of the component. While components promote modularity, for clients it is important to know how to use a component properly. For instance, a client may need to know under which circumstances a certain method invocation may throw an exception. Merely knowing Java's type signature of the method can be insufficient, as correct usage may depend upon the dynamic state of the concerned objects.

This problem can be addressed to a certain extent by associating each object with a typestate [17]. A typestate is an abstraction of the dynamic state of an object. It depends on the problem at hand to figure out which state properties to reflect in the typestate. Typically, only simple properties such as values of flags are included in the typestate. As long as a specification of preconditions and postconditions for each method is given in terms of typestates, a user can use the component properly. However, when inter-object references are present, components can become much more difficult to understand and verify; this is because it is difficult to associate with an object a typestate that encapsulates the relevant aliasing information.

Figure 1 shows two classes `A` and `C`.[1] The code shown contains only statements relevant from the point of view of a client interested in preconditions for calling `A.run()` or `A.set()` without throwing an exception. It is straightforward to describe the relevant behavior of `C` in isolation in terms of typestates. A `C` object is either in the state `C1` in which `C.disposed` is false, or it is in the state `C2` in which `C.disposed` is true. The call to `C.dispose()` changes `C`'s state from `C1` to `C2`, and it is an error to call `C.run()` or `C.dispose()` in state `C2`.

Suppose we attempt to capture the behavior of `A` in terms

[1]This example is an abstracted version of classes `ContentViewer` and `IBaseLabelProvider` in Eclipse [1].

of the typestate of `A.f`. Here, the call to `A.set(C1)` would take `A.f` from `null` to `C1`. One might expect that a subsequent call to `A.set(C1)` would retain the state of `a.f`. Surprisingly, the call sequence `a.set(c)`, `a.set(c)`, `a.run()` leads to an error. This is because the *same* object `c` is passed in as a formal parameter to the second call to `A.set`, and causes `a.f` to point to a `c` object which is now in state `C2`. The call sequence `a.set(c)`, `a'.set(c)`, `a.done()`, `a'.run()` would also lead to an error. Here, the state change caused by the call to `a.done()` ends up changing the state of `a'.f` as well, because the `c` object is shared.[2]

```
class A {
  C f;
  void run() {
    if (f != null) f.run();
  }
  void done() {
    if (f != null) f.dispose();
  }
  void set(C i) {
    if (f != null) f.dispose();
    f = i;
  }
}
class C {
  boolean disposed;
  void run() {
    if (disposed) throw new Error();
  }
  void dispose() {
    if (disposed) throw new Error();
    disposed = true;
  }
}
```

Figure 1: A simple example to show the effect of inter-object references.

As the preceding example shows, in order to characterize the behavior of objects related by inter-object references, typestates and transitions thereon need to factor in the aliasing relationships that hold among the objects involved in a method call in order to prevent these types of errors. Traditional typestate systems do not satisfy this requirement. This paper presents an approach to deriving relevant object typestates in the presence of inter-object references that do satisfy this requirement.

We use static analysis to automatically derive object typestates, accounting both for inter-object references and the local object state. Predicates of interest with respect to the typestates are discovered by analyzing reachability of certain program points, e.g. `throw` statements, within a method. The typestate of the object on which a method is invoked—along with typestates of any objects passed as parameters and possibly values of certain boolean flags—gives the reachability precondition for the method: whether the specified program point *may* be reached. Our analysis also derives transitions between object typestates caused by a method call, taking into account the typestate changes in related objects as well. The effect of a method on an object a may not only update a's typestate, it may update typestates of objects that are referred to by a, and it may also update typestates of any objects passed in as parameters to the call. It may even update typestate of objects that are not reachable from a, but nevertheless have some aliasing relationship of consequence.

A key difficulty in computing and presenting this information is that the client of the component being analyzed is not known *a priori*. The analysis has to somehow account for all possible actions the client code can take that may lead to different possible transitions. We do this analysis by exploring method calls on a *synthesized* heap. It is synthesized in the sense that it does not come from an actual client; it is produced by a client emulator that constructs all heap states of interest. Effects of method calls are expressed in how objects in a heap undergo typestate transitions.

Our overall approach is sketched in Figure 2.

1. We first discover predicates that determine reachability of specified program points in the component. These predicates control, either directly or indirectly, the outcomes of conditionals leading from the component boundary to the specified program points. Therefore, the predicate discovery phase starts with predicates from branches leading to the critical program points. It then performs a limited iterative weakest pre-condition computation which generates additional predicates. The limitations imposed ensure that only a finite number of predicates can be generated. Predicate discovery is performed via backwards intra- and inter-procedural "slicing"; a preliminary inter-procedural dependence analysis is performed to enable this slicing. Because the client code is unknown, the propagation simulates a non-deterministic client that obeys Java type rules to "stitch" the control-flow graphs at the component boundary.

2. The predicates resulting from the predicate discovery are then used for predicate abstraction, in which we convert the code for each method in the input Java classes to "boolean" code. The boolean code abstracts out all scalar computation in the method to reads and writes of predicates of interest. Departing from a truly boolean program, the algorithm retains assignments to reference variables (including reference fields), and allocations in the "boolean" program. Predicates in our boolean code are parameterized over such reference variables. Our boolean abstraction is conservative primarily due to our inability in handling arithmetic in the weakest pre-condition computation, not due to approximations in the preliminary pointer analysis.

3. Given the boolean methods, client emulation and abstract interpretation are used to explore the possible behaviors of the component. Since we have to bound the number of observable behaviors, we map the "states" of the synthetic heap into an *abstract* heap state space, which is finite by design. The abstract heap states bound the number of distinct typestates for any object, as well as the number of distinct typestate transitions that a method can cause. The abstract heap state space is constructed by repeated application of boolean methods over the synthetic heap followed by an abstraction of the synthetic heap. The exploration increases the size of the synthesized heap, up to the point where no new typestates or transitions in the abstract heap state space are observed.

The result of our analysis is a set of object typestates and transitions that describe how a method call influences the typestates of various objects that participate in it. For some

[2]A convention in Eclipse is that `ContentViewer`s cannot share an `IBaseLabelProvider` object, but the API itself does not make this obvious. Also in other situations, certain objects are *supposed* to be shared for efficient utilization of resources.

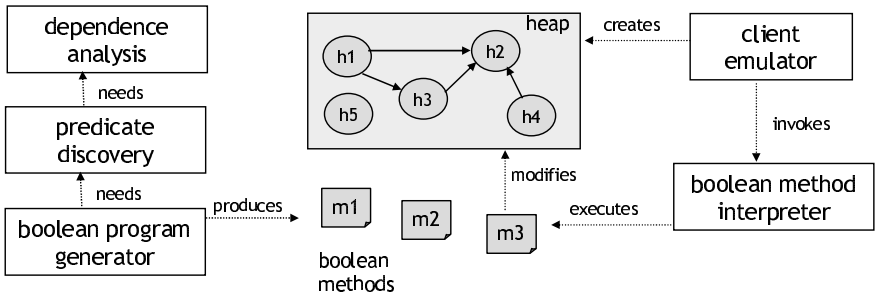


Figure 2: Overview of the approach

methods, some of the transitions are error transitions—that is, the initial typestates of participating objects (or some of the formal parameters) are such that the execution of the method may reach an error statement. The result characterizes the preconditions for *safe* use of a component: as long as the transition followed is not an error transition, the call is guaranteed to not reach an error-causing statement in the corresponding Java method. Due to the limited precision of the boolean abstraction, the analysis may be overly pessimistic, in that it may flag certain safe uses as possibly erroneous. Note that in order to be able to determine legality of calls for a sequence of method calls clients need to maintain typestate information for the participating objects.

Returning to the example shown in Figure 1, our analysis will associate a typestate with `A` that contains information whether `A.f` is in state `C1` or `C2`. The abstract heap state space will show a transition for method call `A.set(C1)`, in which, if the instance of `C` passed in as the formal parameter is aliased by `A.f`, the final typestate of that `C` object will be `C2`; correspondingly the state of `A.f` will be `C2` and a call `A.run()` would be an error transition. The abstract heap state space will also include a transition for the call `A.set(C1)`, in which an unaliased instance of `C` is supplied. In this case, a subsequent call to `A.run` will not be an error transition.

The paper uses two additional examples as running examples to illustrate the various parts of our analysis. The `Set-Iterator` example in Figure 3 shows the use of a Java collection class. We invite the reader to verify that the sequence of calls `i = s.iterator()`, `s.add()`, `i.next()` leads to an error. The `List` example in Figure 4 shows a recursive data structure. Although our analysis is not designed for reasoning about shape properties, we include this example to demonstrate how we finitize object typestates.

We believe that our technique of reachability analysis is a new and powerful static analysis tool; being able to decide reachability is a useful building block for other applications. For example, it may be employed for interface synthesis, program verification and also various optimization problems. Note that the reachability problem can also model the question of whether any user-specified predicate holds after the execution of each method call. Describing these applications in detail is outside the scope of this paper. Specifically, the paper does not include an algorithm for statically verifying client code, though Section 5.6 shows some examples of how this may be done.

The paper is closely related to several threads of recent work. It is related to work on object typestate formalisms, in which manually specified typestates capture aliasing in certain ways [8, 11]. This work was inspired by algorithms that discover typestates for the purpose of interface synthesis [2, 18]. It is also related to work on component verification using predicate abstraction [15] and shape analysis [12, 16]. Related work is discussed in more detail in Section 7.

```
class Set {
  Version ver;
  Object[] contents;
  Set() {
    contents = new Object[0];
    ver = new Version();
  }
  public Iterator iterator() {
    return new Iterator(this);
  }
  public void add(Object o) {
    Object[] tmp = new Object[contents.length+1];
    for (int i=0;i<contents.length;i++)
      tmp[i] = contents[i];
    tmp[contents.length] = o;
    contents = tmp;
    ver = new Version();
  }
}
class Iterator {
  final Version ts;
  final Set set;
  int pos;
  Iterator(Set s) {
    this.ts = s.ver;
    this.set = s;
  }
  public Object next() {
    if (ts != set.ver)
      throw new ConcurrentModificationException();
    return set.contents[pos++];
  }
}
```

Figure 3: `Set` and an `Iterator` over the `Set`

Our overall approach is built upon fairly standard analysis techniques: alias analysis, predicate abstraction, and finite heap abstraction. Our contribution is primarily in adapting a combination of these techniques for a new application. The two most important technical novelties of this paper are:

- We infer relevant object typestates in the presence of inter-object references automatically. In contrast, previous work has either required manual specification of typestates, has restricted aliasing in certain ways, or has ignored inter-object references completely.

```
public class List {
  private Element head;
  List () {
    head = null;
  }
  public void append(int data) {
    Element el = new Element(data);
    add(el);
  }
  private void add(Element el) {
    if ( head == null ) {
      head = el;
      return;
    }
    Element t = head;
    while ( t.next != null ) {
      t = t.next;
    }
    t.next = el;
  }
  public int remove () {
    if ( head == null )
      throw new TestError();
    Element t = head;
    head = head.next;
    return t.data;
  }
} // end of List

class Element {
  int data;
  Element next;
  Element( int data ) {
    this.data = data;
    next = null;
  }
} // end of Element
```

Figure 4: An example to illustrate a recursive data structure

- Moreover, we show how to perform this analysis without client code being given. We use a novel technique of client emulation to determine all interesting object typestates and transitions.

The paper is structured as follows. Section 2 describes preparatory pointer and escape analysis that is subsequently needed for predicate generation. Section 3 describes the predicate abstraction, including a description of interprocedural slicing. Section 4 describes the boolean program model. Section 5 describes how we construct an abstract heap state space via client emulation; it also discusses several examples to highlight interesting aspects of our analysis. Section 6 reports some experimental data. Section 7 describes related work, and Section 8 concludes the paper.

2. PREPARATORY ANALYSIS

This section describes preparatory dependence analysis that must be done on the input Java code to enable predicate generation. The dependence analysis uses a standard iterative fixed-point computation [14, 5]. The purpose of this section is to present the basic program representation, some terminology and the results of this preparatory analysis.

We begin the analysis with an intermediate representation of Java programs that contains a control-flow graph representation of the bytecode for each method. In order to simplify the presentation, we ignore most of the object-oriented features of Java, such as subtyping. Adding support for these basic object-oriented features presents no inherent difficulty for our technique. We assume virtual function calls have been converted to static calls and consider exceptions only to the extent that `throw` statements are considered indicative of errors that need to be prevented.

Terminology. A *variable* is a formal parameter, a local temporary, or a field of a local or formal. In the bytecode representation, multi-level dereferences have already been normalized to single-level dereferences by the introduction of additional local temporaries. A `FormalIn` variable is a formal parameter of a method. An `EscapeIn` variable is an upwardly exposed field of a `FormalIn` or an `EscapeIn`. Later in this section, we also define `FormalOut` and `EscapeOut` variables.

We compute a finite set of symbolic heap *locations* for each method. Every reference variable in the method may point to one or more of these locations at each program point. Locations may represent multiple heap objects. Heap locations can be obtained in three ways inside of a method body:

1. Via references passed as formal arguments (`FormalIn`, labeled `F`i for the i^{th} parameter). By convention, the `this` reference appears implicitly at the beginning of the formal parameter list, and so `F1` is the self object to which `this` points to.
2. Via dereferencing operations, reading reference-typed fields (`EscapeIn`, labeled `E`i, numbered using a counter).
3. Via allocation of objects inside of the method (labeled `O`i, where i is a program-point-specific index). We use the customary approximation that each allocation site always gives the same site-unique heap location.

All locations that are accessible outside the method are termed *escape* locations. All `F` and `E` locations escape by definition. Some `O` locations may also escape, while others may remain local.

Intraprocedural Analysis

There are five basic statements that affect intraprocedural pointer and escape analysis: (i) `(New)`, t_i := `new X`, (ii) `(Assign)`, $t_i := t_j$, (iii) `(PutField)`, $t_i.f := t_j$, (iv) `(GetField)`, $t_i := t_j.f$, and (v) `return` t_i. The format of each statement is summarized in Table 1, as are the results of the dependence analysis. We use the notation `u` $\rightarrow$ `loc` to mean `u` points-to `loc`. We say `u` $\rightarrow$ {`L`} to mean `u` may point-to any location in the set `L`. In the course of our analysis, we also generate `FormalIn`, `EscapeIn`, `FormalOut` and `EscapeOut` statements as explained below.

- At method entry, we create a `FormalIn` statement for each formal parameter and an `EscapeIn` statement for each `EscapeIn` variable as shown in Table 1.
- At a `New` statement `u := new X`, we generate a new heap location `O`i and set `u` $\rightarrow$ `O`i.
- At an `Assign` statement `u := v`, if `v` may point to a set of locations {`L`}, then we set `u` $\rightarrow$ {`L`} at this program point.

- At a (primitive) `GetField` statement `p := v.m`, if `v` points to a set of locations {L}, then for each location `loc` ∈ {L}, we generate a *use* of a variable `loc.m`. Likewise, for a (reference) `GetField` statement `u := v.f`, we generate `loc.f` variables; in addition, we determine the set {L′} of locations that `u` may point to. An upwards-exposed use of `loc.m` or `loc.f` results in the generation of `EscapeIn` statements.
- At a (primitive) `PutField` statement `u.m := q`, if `u` points to a set of locations {L}, then for each location `loc` ∈ {L}, we generate a *definition* of a variable `loc.m`. Likewise, for a (reference) `PutField` statement `u.f := v`, we generate `loc.f` variables; in addition we determine the set {L′} of locations that each `loc.f` variable may point to.
- A (primitive) return statement `return q` is treated as an assignment to a special variable `R`. A (reference) return statement `return v` is treated similarly; we also find the set of locations such that `R` → {L}.
- Def-use information is computed for all local variables as well as field variables `loc.f` and `loc.m`.

Terminology. An `EscapeOut` field is a field of the form `loc.f` or `loc.m`, whose definition is downwardly exposed and `loc` escapes the method. An `O` location may escape in one of two ways:

- if in the definition of an `EscapeOut` field, the rhs is a reference variable that points to that `O` location
- if the return variable may point to that `O` location.

An `EscapeOut` statement is a place-holder for `EscapeOut` fields, placed at the end of the method. A `FormalOut` statement is a place-holder for the return value.

The `FormalIn`, `EscapeIn`, `FormalOut` and `EscapeOut` statements encapsulate the dependence analysis for the method. Table 1 summarizes the results of our dependence analysis.

Example. *Figure 5 shows the bytecode before and after dependence analysis for the* `List` *example. For the moment ignore the columns labeled "Boolean code" and "Executable code". At statement* `a6`*,* `t1.hd` *is used before it is defined. The pointer analysis resolves* `t1.hd` *to* `F1.hd` *and the escape analysis generates the corresponding* `EscapeIn` *for* `F1.hd` *(at statement* `a3`*). At statement* `a9`*,* `t4` *points to* `E1` *along the forward path. An* `EscapeIn`*,* `E2`*, is generated for* `E1.next`*. Along the backedge,* `t4` *points to* `E2` *and hence we generate "may" points-to information. We assume that* `E2.next` *points to* `E2` *and no further* `EscapeIn` *locations are generated. At statement* `a13` *an assignment of* `F1.hd` *generates a corresponding* `EscapeOut` *at* `a15`*.*

Recursive Data Structures. The example above also shows how we handle recursive data structures. We use a 1-limited scheme for "folding" recursive data structures, wherein at a given program point, a reference-type field causes escape-in of a distinct heap object at most once. That is, during the analysis, subsequent dereferences of the same field at the same program point are assumed to point to the same heap object. Thus `E1.next` points to `E2` and `E2.next` also points to `E2`.

Parameter Aliasing. The points-to information generated may vary if two input parameters are aliased. However, since we do not have access to client code, we have no way of determining whether two parameters may be aliased or not. Our analysis conservatively assumes that any two `FormalIn` or `EscapeIn` parameters that are of the same Java type may be aliased. For example, in Figure 5, the analysis would assume that `E1` and `E2` may point to the same heap location.

Interprocedural Analysis

The purpose of interprocedural analysis is to provide linkages from the calling context to a called method. Each call site specifies a list of actual parameters and optionally one return variable to be assigned the value returned from a call. All expressions and field variables are assigned to a local variable before being passed into the call.

Terminology. An `ActualIn` variable is a local variable in the caller that is passed as a parameter to a call. An `ActualIn` statement is place-holder for an `ActualIn` variable. We generate one `ActualIn` statement for each `FormalIn` and `EscapeIn` variable in the callee. An `ActualOut` variable is a local variable in the caller that "catches" the value of a `FormalOut` or an `EscapeOut` variable in the callee. We generate an `ActualOut` statement as a place-holder for each `ActualOut` variable.

We replace the `invoke` statement by a series of `ActualIn` statements followed by a series of `ActualOut` statements as shown in Table 1. We create a `parameter-in` edge from each `ActualIn` node to the corresponding `FormalIn` or `EscapeIn` node and a `parameter-out` edge from each `FormalOut` or `EscapeOut` node to the corresponding `ActualOut` node. (Note that if an `ActualIn` variable represents a recursive location, then this must be reflected in the `ActualOut` generated.)

Example. *Figure 6 shows the analysis of the* `Set-Iterator` *example. The function* `Iterator.<init>` *has two* `FormalIn` *parameters and one* `EscapeIn` *parameter (see statements* `C2, C3, C4`*). In* `Set.Iterator`*, the call at* `D9` *to* `Iterator.<init>` *passes in the* `ActualIn` *parameters* `t2` *and* `t1` *for which we create the* `ActualIn` *statements* `D5` *and* `D6`*. In addition we create a* `GetField` *statement* `t3 := t1.ver` *and pass* `t3` *in as an* `ActualIn` *parameter at* `D8`*.* `Iterator.<init>` *has two* `EscapeOut` *parameters* `F1.set` *and* `F1.ts`*. For each* `EscapeOut` *in the callee, we create two corresponding statements in the caller as follows. At* `C9` *in the callee,* `F1.set` *points-to* `F2`*.* `F1` *and* `F2` *in the callee map to* `t2` *and* `t1` *respectively in the caller. Hence, in the caller, at* `D10` *we create an* `ActualOut` *statement where a local variable (*`t4`*) is assigned* `t1` *which is "returned" by the* `EscapeOut`*. The* `PutField` *at* `D11`*,* `t2.set := t4`*, maps the assignment to* `F1.set` *in the callee appropriately in the caller. Similarly,* `D12` *and* `D13` *are generated to map the* `F1.ts EscapeOut`*.*

3. PREDICATE ABSTRACTION

This section describes the translation of Java programs into boolean programs with reference variables. In order to

Java statement	bytecode			post-analysis			
	statement	DEF	USE	statement	DEF	USE	PTS
FormalIn(prim) FormalIn(ref)				$t_i :=$ $t_i :=$	t_i t_i		 $t_i \rightarrow \mathtt{F}_i$
EscapeIn(prim) EscapeIn(ref)				$\{\mathtt{F}_i \mid \mathtt{E}_i\}.m :=$ $\{\mathtt{F}_i \mid \mathtt{E}_i\}.f :=$	$\{\mathtt{F}_i \mid \mathtt{E}_i\}.m$ $\{\mathtt{F}_i \mid \mathtt{E}_i\}.f$		 $\{\mathtt{F}_i \mid \mathtt{E}_i\}.f \rightarrow \mathtt{E}_j$
New	u := new X	u		u := $\mathtt{O}_i$	u		u $\rightarrow \mathtt{O}_i$
Assign(prim) Assign(ref)	p := q u := v	p u	q v	p := q u := v	p u	q v	 u $\rightarrow$ {L}
GetField(prim) GetField(ref)	p := v.m u := v.f	p u	v.m, v v.f, v	p := {L}.m u := {L}.f	p u	{L}.m, v {L}.f, v	 u $\rightarrow$ {L$'$}
PutField(prim) PutField(ref)	u.m := q u.f := v	u.m u.f	q, u v, u	{L}.m := q {L}.f := v	{L}.m {L}.f	q, u v, u	 {L}.$f \rightarrow$ {L$'$}
Return(void) Return(prim) Return(ref)	return return q return v		 q v	return R := q R := v	 R R	 q v	 R $\rightarrow$ {L}
FormalOut(prim) FormalOut(ref)				:= R := R		R R	 R $\rightarrow$ {L}
EscapeOut(prim) EscapeOut(ref)				:= loc.m := loc.f		loc.m loc.f	 loc.$f \rightarrow$ {L}
Invoke	t_r := call fn($t_1, \ldots, t_n$)	t_r	$t_1, \ldots, t_n$	$ai_1 := t_1$ $ai_n := t_n$ $:= ai_1$ $:= ai_n$ call fn $t_r :=$ $ao_1 :=$ $ao_n :=$	ai_1 ai_n t_r ao_1 ao_n	t_1 t_n ai_1 ai_n	

Table 1: Dependence analysis summarized. u and v are reference variables; p and q are primitive variables; f is a reference field; m is a primitive field.

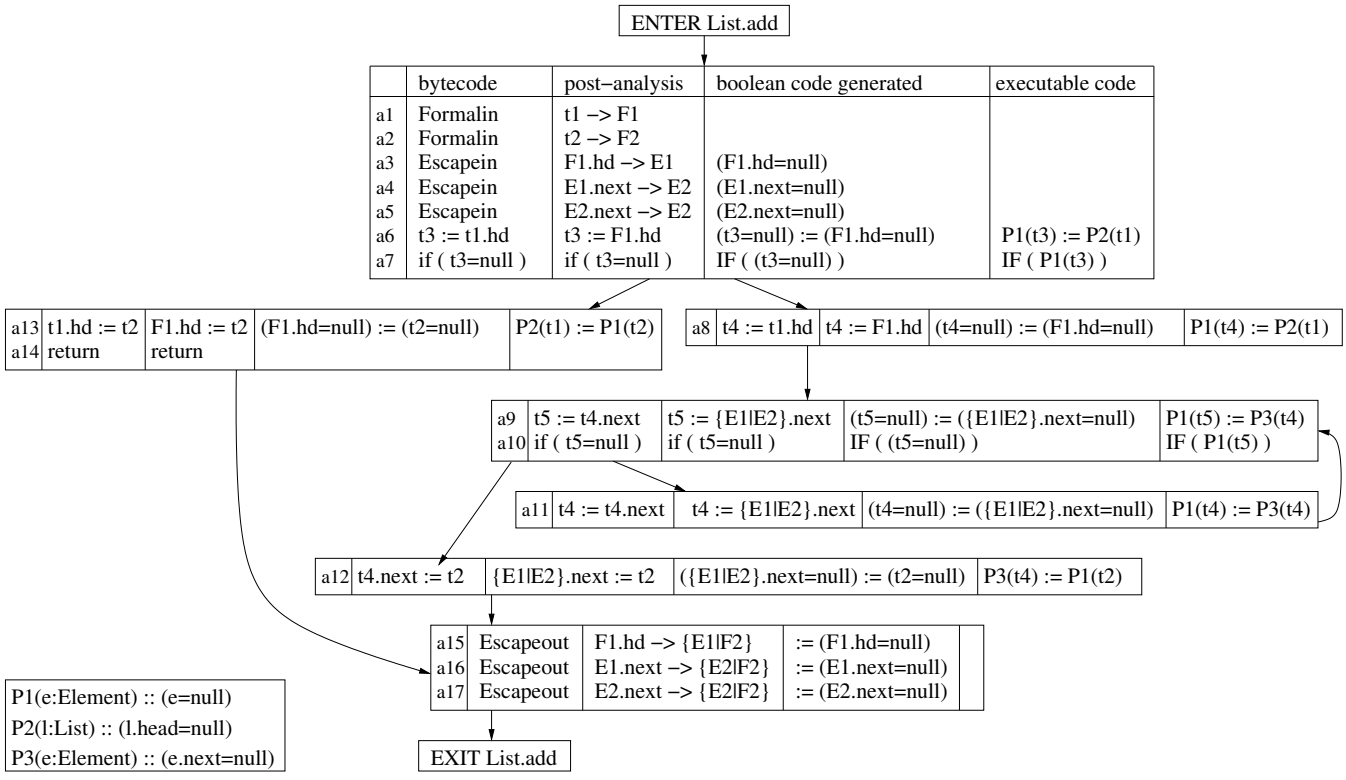


Figure 5: Dependence analysis and boolean code generation for List.add.

perform predicate abstraction, the algorithm needs to discover boolean-valued predicates that encode properties on the state of a Java program that are relevant to the reachability question at hand. The resulting boolean program will then consist of two kinds of operations. First, there are operations on predicates. Secondly, the program contains operations on references, such as creation of objects, passing references as arguments and storing of references in objects. Variables and statements from the Java program that do not correspond to predicate or (relevant) reference operations are deliberately discarded during this translation.

3.1 Predicates

Syntactically, predicates are conditional expressions of the form (*variable op variable*) and (*variable op const*). These expressions evaluate to `true`, `false`, or an indeterminate value `*` which means the predicate could be either `true` or `false`. Furthermore, a predicate's value may be undefined (`?`) which means that the predicate cannot be evaluated (for example, `(a.f=42)` is undefined if `a` is `null`).

Predicate Types

We classify predicates into types where the type of a predicate is defined by the Java types of the variables and the operator. For example `(int < 20)` is a predicate type and `(i < 20)` and `(j < 20)` are instances of this predicate type. Furthermore, for an integer field variable `a.i` where `a` is an object of Java type `A`, the predicate `(a.i < 20)` has the type `(A.i < 20)`; and this type is different from the type `(int < 20)`.

We limit the predicates to the following set of syntactic forms

	Template name	Syntactic structure
1.	`prim-const`	(*prim op const*)
2.	`ref-null`	(*ref* = `null`)
3.	`prim-prim`	(*prim op prim*)
4.	`ref-ref`	(*ref* = *ref*)

In the list above, *op* is one of the operators $<$, $\leq$ or $=$; *prim* is a variable or field of primitive type and *ref* is a reference variable or a reference field. For example, the predicate type `(int < 20)` is of the template `prim-const`, and so is the predicate type `(int = 30)`. Likewise the predicate type `(A.f=B.g)` is of the template `ref-ref`, assuming `f` and `g` are references. Note that the predicate structures are normalized to the above forms. For example, `(int < 20)` is the normalized form for `(20 > int)` as well as for `!(int >= 20)`. This may require exchanging the true and false targets of a conditional in the control-flow graph. A structure `(A.f=B.g)` is the same as `(B.g=A.f)` after normalization. We assign a unique sequence number to each distinct predicate type; this numbering is used to refer to predicates in the generated boolean code.

Terminology. The predicate types derived from the templates `prim-const` and `ref-null` are called *single-variable* predicates, and the predicate types derived from the templates `prim-prim` and `ref-ref` are called *dual-variable* predicates.

Executable predicates

For predicates involving field variables, predicate discovery happens using an intermediate form where all field accesses are of the form `loc.f` where `loc` ranges over possibly many different locations. While the `loc.f` form is suitable for dependence propagation, it is not suitable for precise interpretation of the boolean program, which needs to be able to distinguish between the different heap objects.

For example, in Figure 5 the analysis maybe interested in a predicate `(E2.next=null)` at statement `a9`. During dependence analysis `E2` represents any element in the linked list and consequently `E2.next` always returns the same location, `E2`. For abstract interpretation, these location-based predicates are converted into *executable predicates* which refer to the original variable `t4`. Executable predicates are identified by a predicate type and one or two local variables. For example, at statement `a9`, the boolean predicate `(E2.next=null)` is represented as `P3(t4)`, where `P3` has the type `(Elem.next=null)` and `t4` points to the instance of type `Elem` that the predicate pertains to.

3.2 Predicate Generation

The starting point of the predicate generation phase is a forest of control-flow graphs representing the component functions. We assume that a set of statements in the program has been identified as indicative of an error condition or any condition whose reachability a client may want to control.

Predicate generation starts as follows. For each error-causing statement S, identify c, the conditional expression in the statement S' such that S is control-dependent on S'; c then is a predicate that we need to track in the boolean program. This gives an initial set of predicates to track.

Example. *In Figure 6 the* `throw` *statement at* `A14` *is control dependent on* `A11`. *At* `A11` *we generate the predicate* `(t2=t4)` *of type* `(Version=Version)`, *and the corresponding executable predicate is* `P1(t2,t4)`.

3.2.1 Intraprocedural Propagation

The algorithm presented in this section iteratively discovers predicates by a backwards slicing of the program. The slicing is performed with respect to the variables mentioned in the set of predicates we need to track. At each assignment and conditional statement included in the slice, we generate appropriate boolean code. For each conditional statement encountered in the slice, we generate a boolean conditional statement for the boolean program, and also generate a fresh predicate to be tracked. The treatment of assignment statements is explained below.

Single-variable predicates. Predicate generation is based on computing weakest pre-conditions. Let `(p op const)` be a predicate that is *used* at a certain statement and let the reaching definition of `p` be at S. Then at S, we generate a boolean statement as shown in Table 2. If the defining statement is a `GetField` statement `u` := `v`.f, then to get a complete slice, we also need to find all reaching definitions of `v`. For this reason, we add `v` to the list of slicing criteria. This is the source of reference copies that we generate in the boolean code, which are indicated by `@v` in all figures and tables. All statements in the slice of `v` are added to the "boolean" code and these statements are indicated by

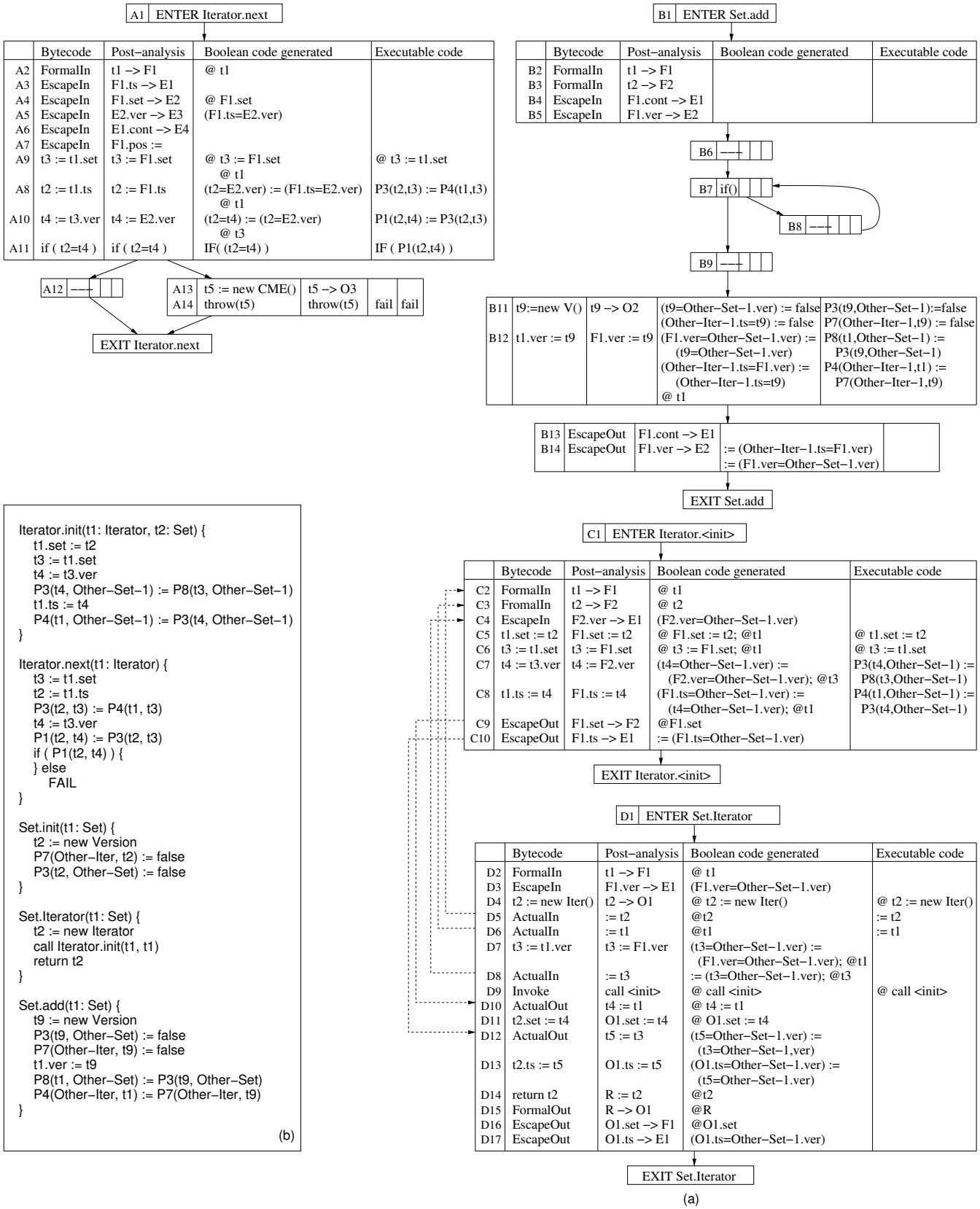


Figure 6: (a) Forest of control-flow graphs for the `Set-Iterator` example (b) Generated boolean program.

	Bytecode	Post−analysis	Boolean code	Executable code
A2	FormalIn	t1 –> F1		
A3	EscapeIn	F1.ts –> E1		
A4	EscapeIn	F1.set –> E2		
A5	EscapeIn	E2.ver –> E3	(F1.ts=E2.ver)	
A8	t2 := t1.ts	t2 := F1.ts	(t2=E2.ver) := (F1.ts=E2.ver)	P3(t2,t3) := P4(t1,t3)
A9	t3 := t1.set	t3 := F1.set		
A10	t4 := t3.ver	t4 := E2.ver	(t2=t4) := (t2=E2.ver)	P1(t2,t4) := P3(t2,t3)
A11	if (t2=t4)	if (t2=t4)	IF((t2=t4))	IF (P1(t2,t4))

Figure 7: Boolean code generation for dual variable predicates

@`statement`. If the defining statement is a `New` statement, `u` := `new T`, equality predicates involving `u` are set to `false`.

We do not attempt to compute precise weakest pre-conditions for all expressions. Instead, we conservatively set the predicate to * if we encounter a complex expression (though in various special cases we can easily determine a definite value without the use of a theorem prover). It is necessary to limit the exactness of the weakest pre-condition computation in order to ensure that the number of predicates generated is finite.

Dual-Variable Predicates. Tracking dependencies in dual variable predicates is similar and the details are summarized in Table 3. Let the predicate `P(u, v)` be a predicate used at a certain statement. Let the reaching definitions of `u` and `v` be at $S_{\mathtt{u}}$ and $S_{\mathtt{v}}$, respectively. Since both definitions reach the same program point, clearly one must post-dominate the other. Without loss of generality, assume $S_{\mathtt{u}}$ post-dominates $S_{\mathtt{v}}$. Then the reaching definition for the predicate is $S_{\mathtt{u}}$ and boolean code is generated according to the rules in Table 3. (Note that the definition of `v` at $S_{\mathtt{v}}$ may still define the *new* predicate generated on the rhs at $S_{\mathtt{u}}$.)

There are certain implementation difficulties associated in tracking dual-reference predicates. Consider an excerpt of code from `Iterator.next` shown in Figure 7. For the predicate `(t2=t4)` at `A11`, the reaching definition of `t2` is at `A8` and the reaching definition of `t4` is at `A10`. Since `A10` post-dominates `A8`, at `A10` we generate the boolean code `(t2=t4)` := `(t2=E2.ver)` and the executable code `P1(t2,t4)` := `P3(t2,t3)`. At `A10`, the reaching definition of `t2` is at `A8` and the reaching definition of `E2.ver` is at `A5`; `A8` post-dominates `A5`. Hence, at `A8` we generate the boolean code `(t2=E2.ver)` := `(F1.ts=E2.ver)`. It is unclear what the executable predicate should be. A candidate executable predicate is `P3(t2,t3)` := `P4(t1,t3)`, but note that at `A8`, the temporary `t3` has not yet been defined, and such code would fail at the time of interpretation. In the executable code, we need to find an appropriate variable that points to the location `E2`. One way to achieve this is to create a temporary copy statement such as $\mathtt{t}_x$`:= t1.set`; insert this statement before `A8` and generate the executable predicate `P3(t2,`$\mathtt{t}_x$`)` := `P4(t1,`$\mathtt{t}_x$`)` at `A8`. Alternatively, we could do the following: we determine that `t3` is defined at `A9`. If placing `A9` before `A8` does not violate any data- or control-dependence, then we reorder the statements and proceed; this is shown in Figure 6. If neither of these options is programmatically feasible, we would have to conservatively generate the code `P3(t2,t3)` := $*$ at `A8`.

3.2.2 Interprocedural Propagation

Interprocedural propagation of predicates can happen both when there is a caller-callee relationship between two methods ("connected" methods), and also when there is not ("disconnected" methods). In the latter case, the dependence comes from the client space, as explained later below.

1. Connected methods: When two functions are connected by a call edge then predicates can be propagated backwards (i) from `ActualOut` nodes in the caller to corresponding `FormalOut` or `EscapeOut` nodes in the callee, and (ii) from `FormalIn` or `EscapeIn` nodes in the callee to corresponding `ActualIn` nodes in the caller. When crossing function boundaries, predicates are rewritten using names that are meaningful in the function body as shown in Figure 8(a). In the case that a dual-variable predicate reaches an `ActualOut` node, if both the predicate variables are defined at the same call site, then both of them are propagated backwards through the callee (Figure 8(b)). If only one of the predicate variables is produced by the callee ((Figure 8(c)), then the *other* variable needs to be propagated with a name that does not pertain to the method body. These variables are termed `OTHER` variables and are expressed generically as `Other-`$\langle$*typeof*$(v)\rangle$*-index*. A different `OTHER` variable is generated for each predicate in a function in which the variable participates.

We maintain a call stack to ensure that predicates are propagated only along the calling context. However, if a function is not in a "called" context, then a predicate that reaches a `FormalIn` / `EscapeIn` statement, is propagate backwards into all calling functions.

2. Disconnected methods: Since our analysis is oblivious to the client program, we need to account for all possible ways in which a client could potentially call the methods in our component. We build interprocedural linkages across disconnected methods as follows: a predicate that reaches a `FormalIn` or `EscapeIn` node when the propagation is not in a called context is said to be visible in the client context. Such a predicate is propagated back into the `FormalOut` or `EscapeOut` node of a function based on type matching, as shown in Figure 8(d). As in the case of connected components, we need to generate `OTHER` variables when a dual-variable predicate is propagated into a function that type-matches only one of the variables (Figure 8(e)).

Example. *In Figure 6* `(F1.ts=E2.ver)` *"escapes upwards" from the* `EscapeIn` *statement* `A5`. *Inspection of* `FormalOut` *and* `EscapeOut` *statements in all the functions shows that* `F1.ver` *escapes downward in* `Set.add` *(statement* `B14`*),* `F1.ts` *escapes downwards in* `Iterator.<init>` *(statement* `C10`*), and* `O1.ts` *escapes downwards in* `Set.Iterator` *(statement* `D16`*). At* `B14`*,* `E2.ver` *is replaced by* `F1.ver` *and* `F1.ts` *is replaced by* `Other-Iterator-1.ts` *to generate the predicate* `(Other-Iterator-1.ts=F1.ver)`. *At* `D16`*,* `F1.ts` *is replaced by* `O1.ts` *and* `E2.ver` *is replaced by* `Other-Set-1.ver` *to generate the predicate* `(O1.ts=Other-Set-1.ver)`. *Hence the predicate is propagated backwards from these statements.*

The complete boolean code generated for the `Set-Iterator` example has been shown in Figure 6(b). Note that we do not generate executable code for `FormalIn`, `EscapeIn`, `FormalOut`, `EscapeOut`, `ActualIn` and `ActualOut` statements. These statements are only used for the purpose of predicate propagation and for parameter matching during execution.

Referenced Predicate/Reference	Defining Statement		Boolean Code Generated	Executable Code
	Bytecode	Post-analysis		
(p op const)	p := const′	p := const′	(p op const) := eval(const′op const)	P_1(p) := {true|false}
	p := q	p := q	(p op const) := (q op const)	P_1(p) := P_1(q)
	p := v.m	p := {L}.m	(p op const) := ({L}.m op const); @v	P_1(p) := P_3(v)
	p := *expr*	p := *expr*	(p op const) := *	P_1(p) := *
(u=null)	u := null	u := null	(u=null) := true	P_2(u) := true
	u := v	u := v	(u=null) := (v=null)	P_2(u) := P_2(v)
	u := new X	u := loc	(u=null) := false	P_2(u) := false
	u := v.f	u := {L}.f	(u=null) := ({L}.f= null); @v	P_2(u) := P_4(v)
(loc.m op const)	u.m := q	{L}.m := q	(loc.m op const) := (q op const); @u	P_3(u) := P_1(q)
	EscapeIn	loc.m :=	(loc.m op const) :=	
(loc.f= null)	u.f := v	{L}.f := v	(loc.f= null) := (v=null); @u	P_4(u) := P_2(v)
	EscapeIn	loc.f :=	(loc.f op const) :=	
@ u	u := null	u := null	@ u := null	@ u := null
	u := v	u := v	@ u := v; @v	@ u := v
	u := new X	u := loc	@ u := new X	@ u := new X
	u := v.f	u := {L}.f	@ u := {L}.f; @v	@ u := v.f

Table 2: Boolean code generation for single-variable predicates. Here op may be either $=$, $<$ or $\leq$. P_1, P_2, P_3, P_4 are predicates of appropriate type.

Referenced Predicate	Defining Statement		Boolean Code Generated	Executable Code
	Bytecode	Post-analysis		
(p op prim)	p := const	p := const	(p op prim) := (const op prim)	P_1(p, prim) := P_2(prim)
	p := q	p := q	(p op prim) := (q op prim)	P_1(p, prim) := P_1(q, prim)
	p := v.m	p := {L}.m	(p op prim) := ({L}.m op prim); @v	P_1(p, prim) := P_3(v, prim)
	p := *expr*	p := *expr*	(p op const) := *	P_1(p, prim) := *
(u=ref)	u := null	u := null	(u=ref) := (null=ref)	P_2(u, ref) := P_5(ref)
	u := v	u := v	(u=ref) := (v=ref)	P_2(u, ref) := P_2(v, ref)
	u := new X	u := loc	(u=ref) := false	P_2(u, ref) := false
	u := v.f	u := {L}.f	(u=ref) := ({L}.f= ref); @v	P_2(u, ref) := P_4(v, ref)
(loc.m op prim)	u.m := q	{L}.m := q	(loc.m op prim) := (q op prim); @u	P_3(u, prim) := P_1(q, prim)
	EscapeIn	loc.m :=	(loc.m op prim) :=	
(loc.f= ref)	u.f := v	{L}.f := v	(loc.f= ref) := (v=ref); @u	P_4(u, ref) := P_2(v, ref)
	EscapeIn	loc.f :=	(loc.f op ref) :=	

Table 3: Boolean code generation for dual-variable predicates. Here op may be either $=$, $<$ or $\leq$. P_1, P_2, P_3, P_4 are predicates of appropriate type. In this table, we assume that the reaching definition of the first variable of the referenced predicate post-dominates the reaching definition of the second variable of the referenced predicate and is found at the defining statement. The reverse situation is symmetric.

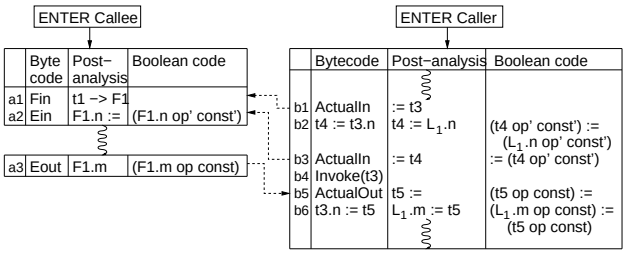
4. BOOLEAN METHODS

The previous phase abstracts a given set of Java classes into a set of abstract procedures. These abstract procedures are similar to boolean programs used ordinarily in predicate abstraction [3], with one important distinction: reference copies present in the unabstracted program are retained in the abstracted program. Unlike customary boolean programs, our abstract procedures contain predicates that are *parameterized* over reference variables. One important reason for this choice is that it allows an abstract interpreter operating on these boolean programs to be more precise than the original points-to analysis that was used to compute the predicates. While the abstracted program in our work is not a "pure" boolean program, for sake of presentation we call it a boolean program in this paper.

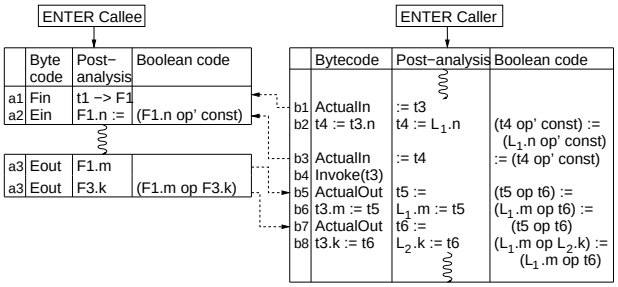
Terminology. A *single-reference* predicate is one in which there is one field lookup involved, i.e. only one occurrence of *refvar.field*, where *refvar* is a reference local variable. A *dual-reference* predicate is one in which there are two occurrences of *refvar.field*. A nullary predicate is one which has no field lookups, i.e. it involves only primitive or reference-typed local variables. A single-reference predicate *pertains* to a type τ if the type of *refvar* on which the field lookup is performed in the predicate is τ. A dual-reference predicate is said to pertain to a type τ if τ is the type of at least one of the two reference variables on which field lookup is performed. A single-reference predicate of the form (*refvar.field op var*), where *var* is a local variable, is said to be a *temporary* predicate, because the predicate is meaningful only in the scope of that local variable; other single-reference predicates are said to be *persistent*, as they exist alongwith the object they pertain to across method calls.

We describe the structure of our boolean programs and salient features of its interpretation. The program consists of the constructs shown in Table 4. Note that the non-reference parameters of a method are abstracted away and instead nullary predicates involving those parameters become part of the parameter list. The same change is also made to the parameter list for call statements inside methods. Additionally, procedures may return a tuple of values, containing possibly several nullary predicate values. Figure 6 gives the boolean code for the `Set-Iterator` example.

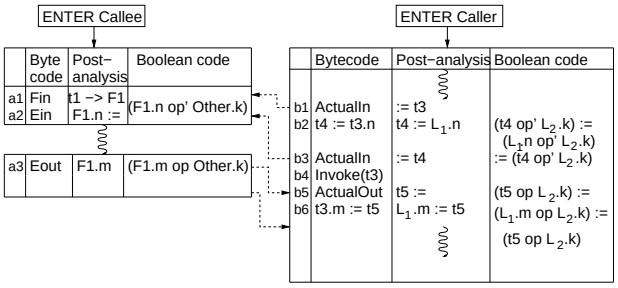
The run-time environment of a boolean program consists of a stack and a heap. The heap consists of a set of typed heap objects, where the types are drawn from the Java types in the component being analyzed. There is no *a priori* limit on the number of objects in the heap. There is also a desig-



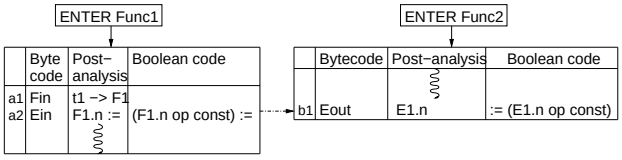
(a) Connected methods: Single−variable predicate propagation



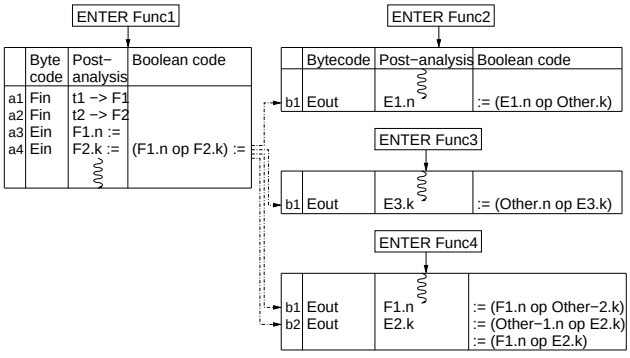
(b) Connected methods: Dual−variable predicate propagation



(c) Connected methods: Dual−variable predicate propagation with OTHER variables



(d)Disconnected methods: Single−variable predicate propagation



(e) Disconnected methods: Dual−variable predicate propagation

Parameter−in/Parameter−out edge
Edge showing matched FormalIn (Fin) / EscapeIn (Ein) nodes with corresponding FormalOut (Fout) / EscapeOut (Eout) nodes

Figure 8: Interprocedural predicate generation.

Construct	*Syntax*
PredAssignStmt	Predicate `:=` Predicate Predicate `:=` BooleanValue
Predicate	*PredId*`(`*refvar*`,` *primvar*`)` *PredId*`(`*refvar*`,` *refvar*`)` *PredId*`(`*refvar*`,` `OTHER)` *PredId*`(`*refvar*`)` *PredId*`()`
BooleanValue	`true` | `false` | `*`
RefCopyStmt	*refvar* `:=` *refvar* *refvar* `:=` *refvar.f* *refvar* `:= null` *refvar.f* `:=` *refvar*
AllocationStmt	*refvar* `:= new X`
ControlFlowStmt	`if (`Predicate`) goto` *l* `else` *l'* `if (`BooleanValue`) goto` *l* `else` *l'* `label` *l*
ProcedureCall	ArgList `:=` *FuncId*`(`ArgList`)` `return` ArgList
ArgList	*refvar* *PredId*`()` *refvar* `,` ArgList *PredId*`()` `,` ArgList
ErrorStmt	`FAIL`

Table 4: Syntax of boolean program. Here *refvar* is a reference local variable, *primvar* is a primitive local variable, *PredId* is a predicate type, and *FuncId* is a method.

nated `null` object, to which all null pointers point. Objects in the heap state may also contain fields that are references to other objects in the heap, exactly as regular Java objects do.

The stack consists of bindings for local variables, which are all of reference type, as the primitive-typed variables have been abstracted away. These variables point to one of the heap objects in the heap. Note that we are primarily interested in how the heap is modified when a method completes its execution. The stack contents are of interest only during the execution of a method.

Execution of a boolean method is set up by assigning `FormalIn` reference parameters from the heap, and assigning true or false values to the other arguments (which are all of predicate type after abstraction). The execution proceeds conventionally in the style of a model checker, following the control flow in the boolean method. When a conditional value evaluates to `*`, either path can be taken. A `new X` statements allocates an unused object of type `X` from the heap; if no such object is available the interpreter terminates abnormally, but without reporting an error. When the execution of a boolean program hits a `FAIL` statement, it terminates with an error.

Next, we describe reads and write of predicates. Conceptually, the run-time environment maintains a store, *PredStore*, of various predicates along with their current values. Nullary predicates are indexed into this store based on the (now abstracted) variable names they contain, e.g. the nullary predicate `P1(x)` is looked up by the tuple $\langle$`P1, x`$\rangle$. Single-reference predicates are indexed based on the heap object to which their *refvar* points to. Thus, the predicate `P3(u,t)`, defined as `(u.f == t)`, is indexed as the tuple

$\langle$P3, o, t$\rangle$, where o is a heap object to which u points, and t is a temporary. Dual-reference predicates are indexed based on both the heap objects their *refvar*s refer to. Two objects that are related by a dual-reference predicate are termed *buddies* with respect to that predicate. Note that a given heap object can have multiple instances of a dual-reference predicate associated with it in *PredStore*, because it can be in a certain relationship with several other heap objects (its buddies w.r.t. a certain predicate). Nullary, and temporary single-reference predicates, are defined only when the current method being executed contains stack bindings of the temporary variables they contain. An interpreter would maintain the information in *PredStore* partly in the heap and partly in the stack, as appropriate for each predicate instance.

Example. *Figure 9 shows four objects:* o_1 *of type* Set *and* o_2, o_3 *and* o_4 *of type* Iterator. *Figure 9(a) shows the situation when* o_1 *has been initialized by an invocation of* Set.init. *The generated predicates are stored in the heap as shown.* o_2, o_3 *and* o_4 *have not yet been initialized, and hence contain no information.*

Figure 9(b) shows the situation just after o_2 *and* o_3 *have been initialized by calls to* Iterator.init(F1:o_2,F2:o_1) *and* Iterator.init(F1:o_3,F2:o_1), *respectively. The predicate* P4(s,i), *defined as* (s.ver == i.ts), *has* $\langle$P4, o_1, $o_2\rangle$ *and* $\langle$P4, o_1, $o_3\rangle$, *both with value* true, *in the* PredStore. o_2 *and* o_3 *are buddies of* o_1 *w.r.t.* P4. *Conversely,* o_1 *is a buddy of* o_2 *and* o_3 *w.r.t.* P4.

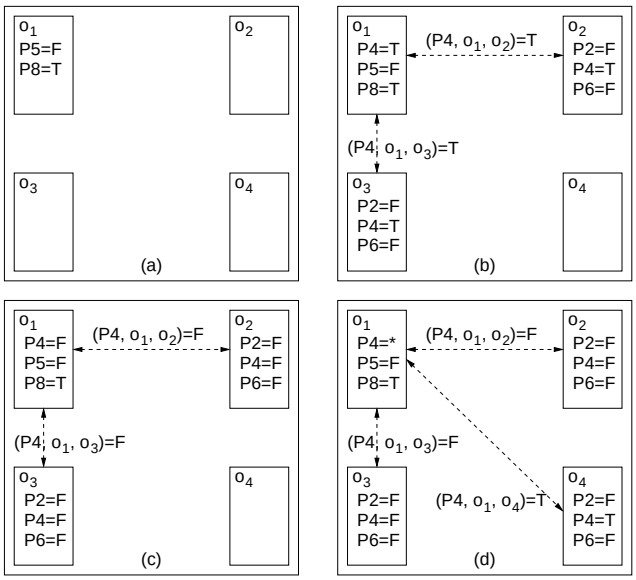


Figure 9: Heap state for the Set-Iterator **example. Object** o_1 **is of type** Set **and objects** o_2, o_3 **and** o_4 **are of type** Iterator.

Predicates with OTHER. Recall from Section 3.2.2 that OTHER can be generated in a dual-reference predicate during interprocedural backward slicing: when the calling context is known (in connected methods) and when the calling context is unknown (in disconnected methods), i.e. the method is called directly from the client. When the calling context is known, the binding of OTHER comes in as a part of the parameter list. However, when the calling context is unknown, we have to find out what it might be bound to. (This distinction in treatment needs to be made at interpretation time.)

For a dual-reference predicate $P(l,$ OTHER-T-1$)$, we look up the heap object referred by l, and then identify all "buddies" it has for the predicate P in the *PredStore*. This gives us a set of bindings for OTHER-T-1. Note that for equality predicates (where the operator is $=$ or $\leq$) where both variables of the predicate are of the same type T.*field*, we consider an implicit binding for OTHER-T-1 to the other location mentioned in the predicate – which makes the predicate by definition true. For an inequality predicate (where the operator is $<$) we consider an implicit binding for OTHER-T-1 in the same way, except the predicate is by definition false.

If both the lhs and the rhs of a statement have an occurrence of the same OTHER variable, then the binding implied by the rhs predicate is used for the lhs occurrence as well. In effect, the statement is executed for each different binding of OTHER. This mechanism thus performs a universal quantification over the set of related objects, and is a feature required to handle predicates on collection classes.

Example. *Continuing the previous example, the statement* P4(s, OTHER-Iter) := false, *where* s *points to* o_1 *would make both* $\langle$P4, o_1, $o_2\rangle$ *and* $\langle$P4, o_1, $o_3\rangle$ false. *This is shown in Figure 9(c).*

5. ABSTRACTING THE HEAP

We now define an *abstract* heap state from the heap of a boolean program's environment. An abstract heap state consists of a set of heap objects in certain *typestates*, as defined shortly. The specific contents of the heap objects are not of significance in the abstract heap state, and neither does the abstract heap state contain references between heap objects. Given a *concrete* heap H, we can apply the typestate computation to each of its constituent objects (ignoring null) to arrive at an *abstract* heap state A. The typestate computation defines an abstraction function α, such that $A = \alpha(H)$.

The key idea behind abstract heap states is that all relevant behaviors of methods, which we need to explore to discover a safe interface, can be found by a *finite* exploration of these abstract heap states. For this purpose, we also require a notion of *transitions* between abstract heap states in response to method calls.

5.1 Typestates

The typestate of a type is the valuation of an ordered set of persistent predicates. The design decision we make here is to include as much information in the typestate as possible in order to be able to reflect the safety precondition of a call. Thus, we not only include the predicates on the receiver object of a method call, but also predicates on any objects that may be accessed by following field-references starting from the receiver object. We think of this as "splitting" typestates into finer distinctions based on additional reachable data.

Example. *In the* A-C *example in Figure 13, the predicates for an object of type* A *are* (A = null) *and* (A.f = null). *These two predicates in isolation are insufficient to distinguish all the different behaviors of* A. *It is important to also include the predicate on the object that* A.f *points to (if not* null*). This lets method calls distinguish behavior of* A *depending on whether* A.f *is disposed or not. Thus the typestate of* A *is defined by the predicates* (A = null), (A.f = null) *and the typestate of* A.f*, as shown in Figure 13(a).*

Two approximations are used in doing this. First, because of the possibility of recursive data structures, we have to take care not to go in cycles collecting an unbounded number of predicates. We bound the number of predicates being collected along any chain of deferences by considering only those field paths that do not repeat a field name. (An alternative strategy would be to bound the number of dereferencing operations on each path.)

Example. *Figure 12(c) gives the typestates for the recursive object* Element *in the* List *example of Figure 4. Consider the linked list of* Elements. o_1 *has the following set of predicates in its state space* {(P1,F), (P3,T), next[]} *which generates the typestate* M1. o_2 *has the following set of predicates in its state space* {(P1,F), (P3,F), next[(P1,F), (P3,T)]} *which generates the typestate* M2. o_3 *has the following set of predicates in its state space* {(P1,F), (P3,F), next[(P1,F), (P3,F)]} *which generates the typestate* M3. o_4 *has the following set of predicates in its state space* {(P1,F), (P3,F), next[(P1,F), (P3,F)]} *which also generates the typestate* M3. *Adding more* Elements *will not generate any further typestates. Note that more than one* next *field is never traversed in typestate computation. The typestates are also summarized in Figure 12(a). The table shows that an object in typestate* M3 *may have a* next *reference that is either in state* M2 *or* M3.

The second approximation pertains to dual-reference predicates. Each dual-reference predicate may be assigned for the same object with several different partners. The typestate computation approximates this information by keeping only three distinct valuations: that the predicate is true with respect to all partners (true), that the predicate is false with respect to all partners (false), and that it is true for some partners and false for others (*).

Example. *Figure 9(d) shows the situation when* o_4 *has been initialized by a call to* Iterator.init(F1:o_4,F2:o_1). *The PredStore now contains* $\langle$P4, o_1, $o_4\rangle$*, with the value* true*, in addition to the existing tuples* $\langle$P4, o_1, $o_2\rangle$ *and* $\langle$P4, o_1, $o_3\rangle$ *both with value* false. *The predicate value of* P4 *in* o_1 *is set to* * *since for some buddies the predicate is true and for some it is false. The predicate value of* P4 *in* o_4 *is set to* true *since for all its buddies (it has a single buddy* o_1*) the predicate is true. Similarly, the predicate value of* P4 *in* o_2 *and* o_3 *is* false. *Note that in Figure 9(b), all the buddies of* o_1 *have a value* true *for* P4 *and so the value of* P4 *in* o_1 *is also* true. *In Figure 9(c), all the buddies of* o_1 *have a value* false *for* P4 *and so the value of* P4 *in* o_1 *is also* false.

Consequently, the typestate for each type consists of a *finite* length string of predicates, and therefore for each type there are a finite number of distinct typestates.

For a type τ, we denote its typestates as τ^0, τ^1, τ^2 etc., where τ^0 is the state of an object on which the constructor has not run yet. Each τ^i is a map $Path \times Pred \rightarrow BVal$, where *Path* is either self or an object reachable via non-repeating field references, *Pred* is a predicate on the object thus reached, and *BVal* is T/F/*. At times, when the concrete heap object passed in as an EscapeIn of type τ is null, we assume its typestate is τ^0.

5.2 Transitions

A *transition* describes typestate changes in heap objects that participate in the various locations (F, E, and O) in a method call. In the presence of dual-reference predicates, each such object can also have buddy objects that too can be impacted by a method because of the potential update to that predicate. It is convenient to assign locations to buddy objects as well: all objects that are related to a particular F, E, or O location via a particular dual-reference predicate belong to the same buddy location (denoted Bi). The locations F, E, O and B together describe the objects that are potentially impacted by a method, in that one or more predicates on those objects may be updated by the method and therefore their typestate can change.

There is also an *indirect* impact on objects that point to any of the objects impacted by a method. We call such objects "parent" objects. The typestate change on a parent object maybe of consequence for verifying client code, but is not important to the transition. (We will come back to the topic of client code verification Section 5.6.)

The correspondence between objects and locations is not one-to-one. In certain cases, a location describes a multiplicity of objects, whose count is not bounded a priori. This occurs in the case of recursive data structures. Recall from Section 2 that a finite number of locations represent all objects constituting a recursive structure due to 1-limiting used in the computation of locations E and O. This also occurs in the case of buddy locations (Bi), since multiple objects can be related to a particular location by a particular dual-reference predicate. In both of these cases, we need to finitize the number of distinct transitions that involve such a "multi" location. The abstraction we use is that a transition contains all possible final typestates for each distinct initial typestate at that "multi" location, instead of a list of per-object typestate changes.

When a single object maps to several locations—a form of parameter aliasing—it can cause distinct transitions, as shown in the example below. In general, aliasing could occur between multiple pairs of locations simultaneously. Since different aliasing combinations might lead to different transitions, each distinct combination must be considered separately to account for all different transitions. Therefore, transitions are annotated with a description of which pairs of locations are aliased.

Example. *In Figure 13(b), both the transitions* T_{17} *and* T_{18} *take as input objects in state* A2, C1 *and* C1 *as* F1, F2 *and* E1 *parameters respectively. However, in* T_{18}*, the* F2 *and* E1 *parameters are aliased. Hence in* F2 *and* E1*, the transition for the* F2 *parameter is* C1 $\rightarrow$ C2 *instead of* C1 $\rightarrow$ C1 *as in* T_{17}*. This may come as a surprise to an unsuspecting programmer, who does not expect the state of the* F2 *parameter to change.*

To summarize, for each location of a method, a transition contains an initial typestate and *a* final typestate.[3] For a "multi" location, a transition contains initial and final transition pairs for all distinct initial typestates participating in that location. It is easy to see that the number of distinct transitions is finite. Formally, a transition for a method is a map $Loc \times TypeState \rightarrow TypeState$ except for "multi" locations, for which it is a map $Loc \times TypeState \rightarrow 2^{TypeState}$. The aliasing context of the transition, when present, is a list of pairs $Loc \times Loc$.

Next, we describe how these transitions relate abstract heap states and help create the abstract heap state space.

5.3 Defining an Abstract Heap State Space

Let $A = \{o_1, o_2, ..., o_n\}$ be an abstract heap state, with objects in certain typestates (not necessarily distinct), and let A' be another abstract heap state in which the typestates of some or all of these objects may have changed. We say that $A' = A(\mu, T)$ if,

- The mapping μ maps (a subset of) objects in A to locations of the transition T;
- The aliasing constraint, if any, in transition T is satisfied, that is, if (l_1, l_2) is a pair in the aliasing constraint of T, then μ maps a common object to both locations;
- For each object o_i participating in the mapping μ, the typestate of o_i in A' is consistent with T, that is, if $\mu(o_i) = l$, typestate of o_i in A is τ^p, and typestate of o_i in A' is τ^q, then $\tau^q = T(l, \tau^p)$, or if l is a multi-location, then $\tau^q \in T(l, \tau^p)$; and,
- The typestate of each object in A not participating in μ is unchanged in A'.

For a set of given boolean methods, and their corresponding sets of transitions (recall that each method can give rise to several transitions), the abstract heap state *space* is a graph whose nodes are abstract heap states, and an edge from abstract heap state A to A' indicates that there exists a mapping μ and a transition T such that $A' = A(\mu, T)$. The graph contains an initial state $A_\perp$, in which all objects are in their respective uninitialized typestates (τ^0). There is also a designated error node, and all error-causing transitions lead to the error node.

Our goal is obtain a *maximal* graph — that is, to add all reachable states and to add all possible edges starting with a given set of objects in state $A_\perp$. Nodes and edges in this graph are added by observing the effects of a method call on a certain concrete heap state; this is described in the next subsection.

One key issue is whether the starting state contains a sufficient number of objects to allows for the observation of all possible behaviors. Our technique systematically tries out initial states with an increasing number of objects, until no new transitions can be found. Recall that the number of different transitions is in fact bounded, and thus an upper bound on the number of objects needed to observe all transitions must exist.

[3]Different final typestates, which may appear for the same initial typestate along different execution paths through a method, are shown in separate transitions.

5.4 Creating the Abstract Heap State Space

Our approach for finding all relevant abstract heap states and transitions is to emulate all possible client actions on a concrete heap and observe the effects. A client can create new objects in the heap, and can call various sequences of methods on heap objects. A client also has flexibility in how it binds heap objects to formal parameters, as long as the types are matched. Since our boolean methods act on a concrete heap, we track client actions on a synthesized concrete heap H, and shadow those changes in the corresponding abstract heap state A. H is a concretization of A, that is $H \in \alpha^{-1}(A)$.

We now describe how we explore the abstract heap state space, starting with a given $A_\perp$. The concrete heap H at this time consists of uninitialized heap objects of the same corresponding types as in the abstract heap state $A_\perp$; H also includes a `null` object.

1. The client emulator selects non-deterministically a method m from the set of methods available to it. Note that only constructor methods may be invoked on uninitialized heap objects, and the constructor may not be invoked subsequently.
2. The client emulator then *binds* non-deterministically the formal parameters (`F` locations) of m from type-appropriate objects in H. Further details on binding are given below.
3. The emulator invokes the interpreter to execute some (possibly cyclic) path in the boolean method m in this environment. The execution can produce the following outcomes:
   - It reaches the exit point of the method. In this case, the heap objects in H are examined for their typestates, and possible new abstract state and a transition is created in the abstract heap state space. The mapping μ induced by the execution of the method is recorded. The aliasing implicit in the mapping of objects to locations is also recorded along with the transition. The emulator also retains the concrete heap H obtained at the exit to enable further exploration.
   - It reaches a `new X` where no unused object of type `X` is available in H. The execution is abandoned at this point, without reporting an error.
   - It reaches an error statement. We record this as well in the abstract state space by an error transition.

The emulator then backtracks, and tries different paths in step 3, different bindings in step 2 and different methods in step 1. For all normal executions, the abstract heap states, transitions, and the final concrete heap are stored. The error transitions are also remembered.

Next, the above procedure is repeated for each of the new abstract heap states, starting with the concrete heap stored alongwith. This continues until no new typestates or transitions are discovered.

Once the abstract heap state space of a given set of objects has been explored, the emulator increases the number of objects in the heap state and repeats the whole process. With more objects, additional behaviors can be observed.

For one thing, certain executions that previously terminated at a `new` statement may run to completion. Also, different aliasing contexts might be exercised. We describe further below the way in which the emulator explores heap states of various sizes, until even increasing the number of objects in the initial heap does not yield any new typestates or transitions.

Setting up the input parameters. The client emulator inspects the types of each reference formal parameter. For each formal parameter, it finds all possible heap objects in H that match based on the type, as well as the `null` object, which matches every type. A heap object of the appropriate type may be available in various typestates in the current heap state – the algorithm tries all of them. In some of these attempts, it may alias formal parameters among themselves, or may alias formal parameters with `EscapeIn` (which are determined automatically once formal parameters are bound). This is intentional, as parameter aliasing may lead to distinct behaviors.

The emulator tries both `true` and `false` values for each `FormalIn` predicate; recall that while formal parameters of primitive type are abstracted away, predicates involving them may still be relevant.

Increasing the Size of Abstract Heap. The emulator starts with an $A_\perp$ in which there is one object of each known type. Once this $A_\perp$ has been fully explored, the emulator tries to increase the size of the heap state, so that more typestates or transitions can be discovered. The emulator adds additional objects guided by the following considerations:

(a) The exploration of abstract heap states must ensure that given any method in the component with `FormalIn` parameters $f_1, \ldots, f_m$, each parameter f_i will be bound with objects in all possible typestates of the type of f_i. In addition, sufficient number of distinct objects in each of those typestates must exist in order to exercise all possible aliasing combinations. Once the current set of typestates for each type is known, the number of objects needed to make this happen can be computed; the emulator adds the required number of objects to the heap state.

(b) A method may contain a certain number of `new` τ_k statements. We determine the number of objects of type τ_k necessary to let the execution of such a method complete along some path. Those objects need to be added in their initial typestates τ_k^0.

After adding the additional objects the emulator is run again. This process repeats until no additional typestates or transitions are discovered.

The abstract heap state space of the largest initial configuration explored in this process has the following properties:

Safety Executing any sequence of non-failing transitions described by the abstract heap state space will not lead to reaching an error statement.

Maximality Adding additional objects will not allow the observation of additional transitions or typestates that are reachable from $A_\perp$ on safe paths.

Minimality It is not possible to remove transitions from the final heap state space without violating *maximality*.

5.5 Examples

Figure 10 shows two different sized abstract heap states for the `Set-Iterator` example. Figure 10(a) shows the set of transitions that may be discovered when the heap state consists of exactly one `Iterator` object (labeled o_1) and one `Set` object (labeled o_2). Figure 10(b) shows the set of transitions that may be discovered when the heap state consists of two `Iterator` objects (labeled o_1 and o_3) and one `Set` object (labeled o_2).

In the initial heap state, all objects are uninitialized and hence in Figure 10(a), o_1 is in typestate `R0` and o_2 is in typestate `S0`. From this heap state, executing the method `Set.init` with the o_2 object mapped to the input parameter `F1` causes the o_2 object to change from a typestate of `S0` to `S1`. This results in a new heap state, as shown. The transition, labeled T_{10}, contains the typestate change {`F1:S0` → `S1`}. From the heap state labeled {`R0, S1`}, executing `Iterator.init` generates the transition, labeled T_2, which contains {`F1:R1` → `R2`, `F2:S1` → `S2`}. The figure shows that a buddy relationship has been discovered and that the predicate value binding the buddies is currently `true`; however, the buddy relationship itself is not a part of the abstract heap state.

Nine distinct transitions are found with these two objects in the `Set-Iterator` example. Increasing the heap state by adding one more `Iterator` (as shown in Figure 10(b)), generates seven additional transitions. The complete set of transitions is shown in Figure 11(b).

Figures 11, 12 and 13 show the typestates and the transitions for the `Set-Iterator`, `List` and `A-C` example respectively. As mentioned earlier, transitions map initial typestates to final typestates for every location that may be impacted by a method call. For example, in Figure 11(b), the method `Iterator.init` effects the `this` parameter, `F1`, the `Set` parameter, `F2`, and buddies of `F2` designated as `B1`. In each case, a transition is defined by a location-wise mapping of initial and final typestates for each object that participates in the method. Each transition contains a possible aliasing context, for example, T_{18} in Figure 13. Where not shown, the aliasing context is inapplicable to the transition.

Per-Object State Transitions. From these transitions, we can extract per-object state transition diagrams which give a more readable view of typestate changes of an object of a certain type under the various method calls. For example, in the `Set-Iterator` example (Figure 11(c)), an object in typestate `R0` will move to a typestate `R1` if initialized by a call to `Iterator.init` with the `F2` parameter in typestate `S1` (transition T_2), but will move to a typestate `R3` if initialized by a call to `Iterator.init` with the `F2` parameter in typestate `S3` (transition T_4). It is interesting to note that an `Iterator` object can change from typestate `R1` to `R2` or from `R2` to `R4` *only* as "buddy" transitions – that is, these typestate changes occur only as side effects to certain method calls and are not visible otherwise.

5.6 Using Abstract Heap State Space for Client Code Checking

In this section we discuss, informally, how the transitions may be used by a client verifier.

Given a set of objects in some specified typestates and a method to be applied to the objects, and optionally an aliasing context, it is possible to determine which transi-

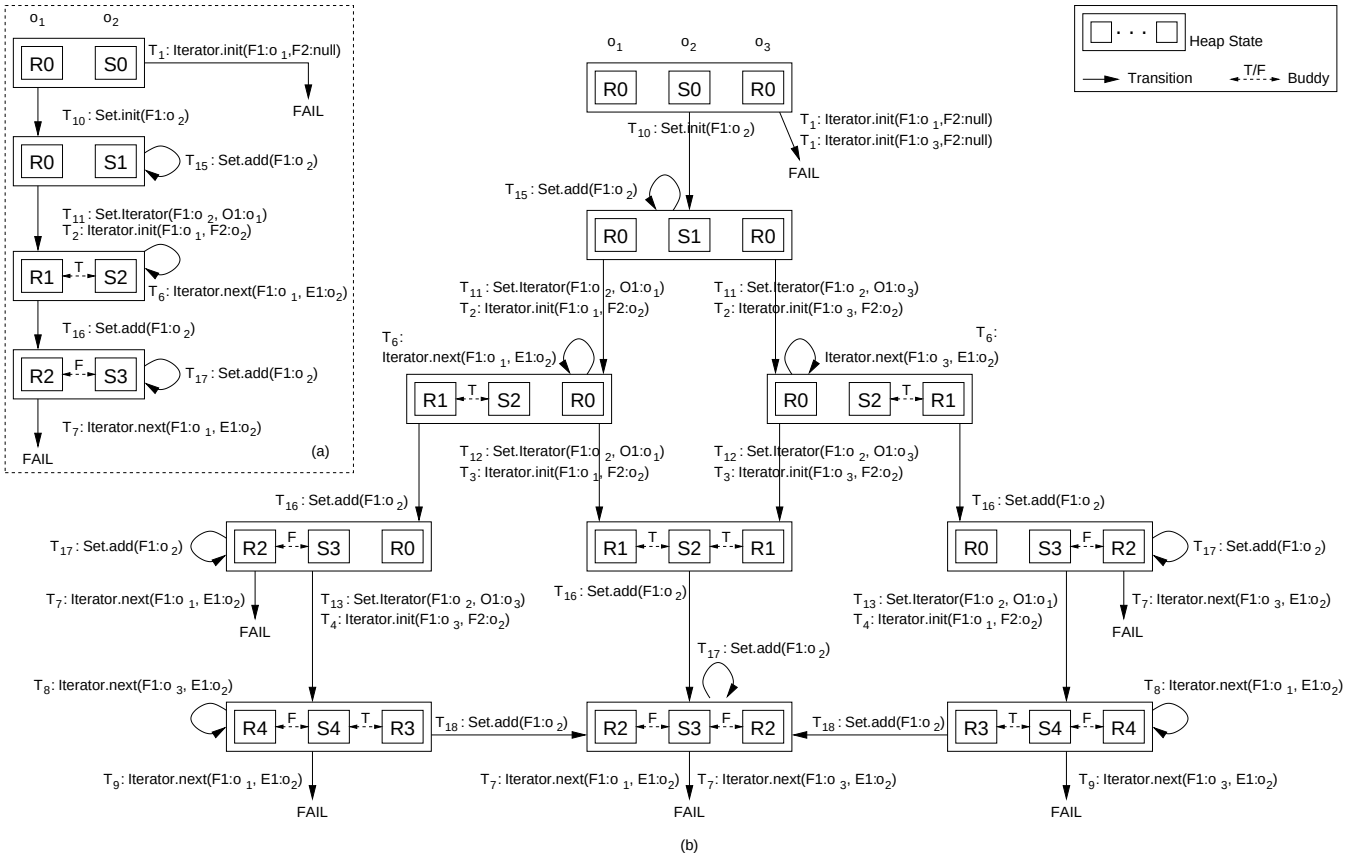


Figure 10: Exploring the heap state space for the `Set-Iterator` example.

tion will apply, and hence the resultant typestates that each participating object will be in. A client verifier "executes" a sequence of method calls by applying a series of transitions as dictated by the client program and the object typestates at the beginning of each step. Consider the `Set-Iterator` client program described in the introduction. We show below how a client verifier would determine that the program will lead to an error.

client code	transition applied	
`s = new Set()`	T_{10}	`s:S0` $\rightarrow$ `S1`
`i = s.Iterator()`	T_{11}	`s:S1` $\rightarrow$ `S2`, `i:R0` $\rightarrow$ `R1`
`s.add()`	T_{16}	`s:S2` $\rightarrow$ `S3`, `i:R1` $\rightarrow$ `R2`
`i.next()`	T_7	`i:R2` $\rightarrow$ `FAIL`

For this example to work correctly, the client verifier needs to keep track of the fact that the call `i = s.Iterator()` puts `t` in a buddy `B1` location of `Set.add()`.

The next example shows how aliasing context can influence the outcome of a sequence of method calls.

client code	transition applied	
`c = new C()`	T_1	`c:C0` $\rightarrow$ `C1`
`a = new A()`	T_6	`a:A0` $\rightarrow$ `A1`
`a.set(c)`	T_{14}	`a:A1` $\rightarrow$ `A2`, `c:C1` $\rightarrow$ `C1`; `a.f` $\rightarrow$ `c`
`a.set(c)`	T_{18}	`a:A2` $\rightarrow$ `A3`, `c:C1` $\rightarrow$ `C2`
`a.done()`	T_{12}	`a:A3` $\rightarrow$ `fail`, `c:C2` $\rightarrow$ `fail`

Here the first call to `a.set(c)`, sets up causes `a.f` to alias `c`. In the second call to `a.set(c)`, the client must use the transition T_{18}—which recognizes aliasing of locations `F2` and `E1`—and not T_{17}.

Parent Transitions. Transitions capture the typestate changes at each location that participates in a method call. They do not capture the *indirect* effect on objects that point to any of the objects participation in a call. The typestate changes of these "parent" objects must be computed by a client verifier. Consider the set of method calls

client code	transition applied	
`c = new C()`	T_1	`c:C0` $\rightarrow$ `C1`
`a = new A()`	T_6	`a:A0` $\rightarrow$ `A1`
`a' = new A()`	T_6	`a':A0` $\rightarrow$ `A1`
`a.set(c)`	T_{14}	`a:A1` $\rightarrow$ `A2`, `c:C1` $\rightarrow$ `C1`; `a.f` $\rightarrow$ `c`
`a'.set(c)`	T_{14}	`a':A1` $\rightarrow$ `A2`, `c:C1` $\rightarrow$ `C1`; `a'.f` $\rightarrow$ `c`
`a.done()`	T_{11}	`a:A2` $\rightarrow$ `A3`, `c:C1` $\rightarrow$ `C2`

The transition T_{11} indicates that `c` goes from `C1` to `C2` and (as a result) `a` goes from `A2` to `A3`. It does not capture the effect of the transition `c:C1` $\rightarrow$ `C2` on the `a'` object, which here is the "parent" of `c`, `a'.f` $\rightarrow$ `c`. A subsequent call to `a'.run()` might assume that `a'` is still in typestate `A2` and erroneously infer that the call will succeed.

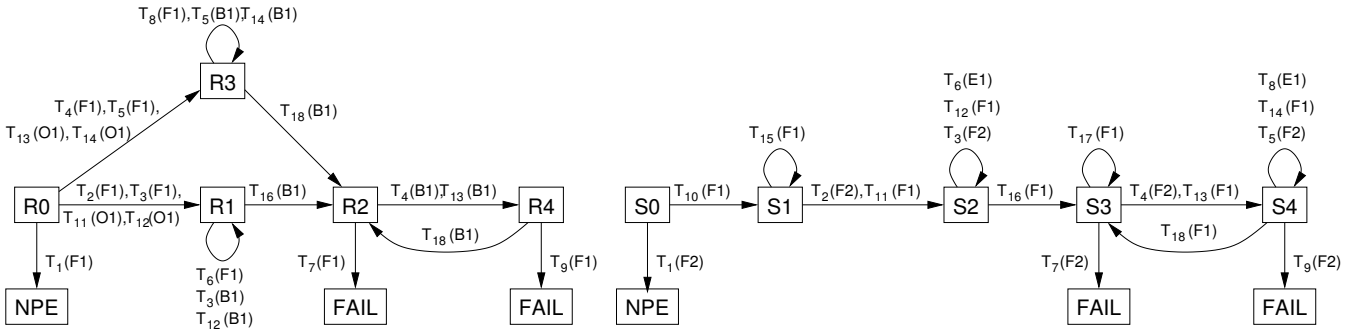
To handle such parent transitions, the client verifier needs to do the following: whenever it accesses an object, it uses the parent-child bindings generated thus far to check if any child (or descendant) object has changed typestate. It must then re-evaluate the parents' current typestate before deter-

τ	pred-value	field-typestate
typestates for objects of type `Set`		
S1	P5 = F P8 = T	
S2	P4 = T P5 = F P8 = T	
S3	P4 = F P5 = F P8 = T	
S4	P4 = * P5 = F P8 = T	
typestates for objects of type `Iterator`		
R1	P2 = F P4 = T P6 = F	%.set → S2
R2	P2 = F P4 = F P6 = F	%.set → S3
R3	P2 = F P4 = T P6 = F	%.set → S4
R4	P2 = F P4 = F P6 = F	%.set → S4

(a) typestates for `Set` and `Iterator`.

Id / function	locations			aliases
`Iterator.init`	F1	F2	B1: P4(?, F2)	
T_1	R0 → ?	S0 → ?		
T_2	R0 → R1	S1 → S2		
T_3	R0 → R1	S2 → S2	R1 → R1	
T_4	R0 → R3	S3 → S4	R2 → R4	
T_5	R0 → R3	S4 → S4	R3 → R3, R4 → R4	
`Iterator.next`	F1	E1: F1.set	B1: P4(?, E1)	
T_6	R1 → R1	S2 → S2	R1 → R1	(F1,B1)
T_7	R2 → ?	S3 → ?	R2 → ?	(F1,B1)
T_8	R3 → R3	S4 → S4	R3 → R3, R4 → R4	(F1,B1)
T_9	R4 → ?	S4 → ?	R3 → ?, R4 → ?	(F1,B1)
`Set.init`	F1			
T_{10}	S0 → S1			
`Set.Iterator`	F1	O1	B1: P4(?, F1)	
T_{11}	S1 → S2	R0 → R1		
T_{12}	S2 → S2	R0 → R1	R1 → R1	
T_{13}	S3 → S4	R0 → R3	R2 → R4	
T_{14}	S4 → S4	R0 → R3	R3 → R3, R4 → R4	
`Set.add`	F1	B1: P4(?, F1)		
T_{15}	S1 → S1			
T_{16}	S2 → S3	R1 → R2		
T_{17}	S3 → S3	R2 → R2		
T_{18}	S4 → S3	R3 → R2, R4 → R2		

(b) Transition maps. Since `B1` is a *multi*-location, the alias `(F1, B1)` indicates that *some* locations in `B1` are aliased to `F1`.



(c) Object State Transitions for `Iterator` and `Set`.

Figure 11: `Set-Iterator` example. Predicate summary: `P2`(r : `Iterator`) :: (r=`null`), `P4`(r : `Iterator`, s : `Set`) :: (r.`ts` = s.`version`), `P5`(s : `Set`) :: (s=`null`), `P6`(r : `Iterator`) :: (r.`set`=`null`), `P8`($s1$: `Set`, $s2$: `Set`) :: ($s1$.`version` = $s2$.`version`).

mining which transition to apply. In the example above, before executing `a'.run()`, the client verifier determines that `a'.f` is now in typestate `C2` and hence `a'` is in typestate `A3` and the applicable transition is T_{12}, which results in failure.

Recursive Data Structures. We need to bound the typestates generated for recursive data structures and this may lead to some loss of information. In the `List` example, starting with an initialized `List` object in typestate `L1`, successive calls to `append` takes the `List` object from `L1` to `L2` to `L3` to `L4` and subsequent calls to `append` does not further change the typestate of the `List` object. From typestate `L4`, there are two transitions T_8 and T_9 on a call to `remove`. When the client verifier finds a situation where more than one possible transition is applicable, it may return conservative results.

6. RESULTS

In this section we briefly describe more examples that we tested our algorithm against, 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 various components of the algorithm.

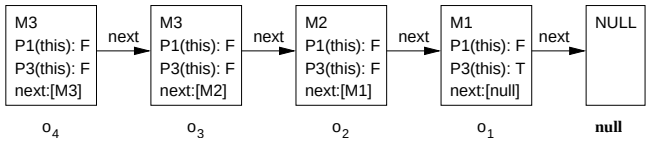
Sample Programs. In addition to the `A-C` example, the `List` example and the `Set-Iterator` example that have been extensively described in the text we ran our implementation on the following test examples: The `Grabbed Resource Problem (GRP)` example described in the Canvas paper [15] throws an error if there is more than one simultaneous traversal of a graph. The `TooManyListeners (TML)` example is an emulation of the `DropTarget` class from

τ	pred-value	field-typestate
typestates for objects of type `Element`		
M1	P1 = F P3 = T	%.next → M0
M2	P1 = F P3 = F	%.next → M1
M3	P1 = F P3 = F	%.next → M2,M3
typestates for objects of type `List`		
L1	P2 = T P4 = F	%.head → M0
L2	P2 = F P4 = F	%.head → M1
L3	P2 = F P4 = F	%.head → M2
L4	P2 = F P4 = F	%.head → M3

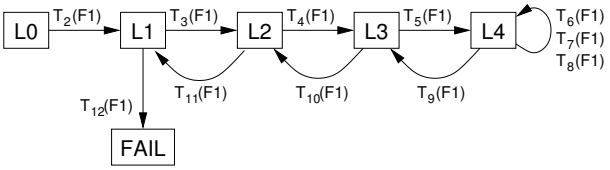
(a) typestates for `Element` and `List`.

Id / function	locations			
Element.init	F1			
T_1	M0 → M1			
List.init	F1			
T_2	L0 → L1			
List.append	F1	O1	E1: F1.hd	E2: F1.hd.next
T_3	L1 → L2	M0 → M1	M0 → M1	→ M0
T_4	L2 → L3	M0 → M1	M1 → M2	M0 → M1
T_5	L3 → L4	M0 → M1	M2 → M3	M0 → M1, M1 → M2
T_6	L4 → L4	M0 → M1	M3 → M3	M0 → M1, M1 → M2, M2 → M3
T_7	L4 → L4	M0 → M1	M3 → M3	M0 → M1, M1 → M2, M2 → M3, M3 → M3
List.remove	F1		E1: F1.hd	E2: F1.hd.next
T_8	L4 → L4		M3 → M3	M3 → M3, M2 → M2, M1 → M1
T_9	L4 → L3		M3 → M3	M2 → M2, M1 → M1
T_{10}	L3 → L2		M2 → M2	M1 → M1
T_{11}	L2 → L1		M1 → M1	
T_{12}	L1 → ?		M0 → ?	

(b) Transition maps.



(c) Typestates for `Element`.

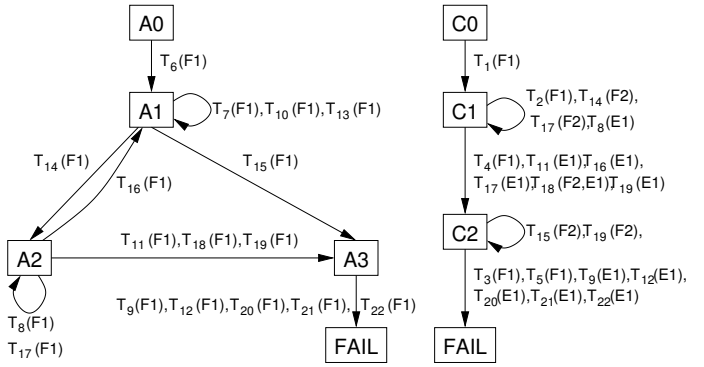


(d) Object State Transitions for `List`.

Figure 12: `List` example. Predicate summary: `P1(`e `: Element)` :: (e`=null)`, `P2(`l `: List)` :: (l`.head=null)`, `P3(`e `: Element)` :: (e`.next=null)`, `P4(`l `: List)` :: (l`=null)`.

τ	pred-value	field-typestate
typestates for objects of type `C`		
C1	P2 = T P3 = F	
C2	P2 = F P3 = F	
typestates for objects of type `A`		
A1	P4 = T P5 = F	%.f → C0
A2	P4 = F P5 = F	%.f → C1
A3	P4 = F P5 = F	%.f → C2

(a) typestates for `C` and `A`.



(b) Object State Transitions for `A` and `C`.

Id / function	locations			aliases
C.init	F1			
T_1	C0 → C1			
C.run	F1			
T_2	C1 → C1			
T_3	C2 → ?			
C.dispose	F1			
T_4	C1 → C2			
T_5	C2 → ?			
A.init	F1			
T_6	A0 → A1			
A.run	F1	E1: F1.f		
T_7	A1 → A1	C0 → C0		
T_8	A2 → A2	C1 → C1		
T_9	A3 → ?	C2 → ?		
A.done	F1	E1: F1.f		
T_{10}	A1 → A1	C0 → C0		
T_{11}	A2 → A3	C1 → C2		
T_{12}	A3 → ?	C2 → ?		
A.set	F1	F2	E1: F1.f	
T_{13}	A1 → A1	C0 → C0	C0 → C0	
T_{14}	A1 → A2	C1 → C1	C0 → C0	
T_{15}	A1 → A3	C2 → C2	C0 → C0	
T_{16}	A2 → A1	C0 → C0	C1 → C2	
T_{17}	A2 → A2	C1 → C1	C1 → C2	
T_{18}	A2 → A3	C1 → C2	C1 → C2	(F2,E1)
T_{19}	A2 → A3	C2 → C2	C1 → C2	
T_{20}	A3 → ?	C0 → ?	C2 → ?	
T_{21}	A3 → ?	C1 → ?	C2 → ?	
T_{22}	A3 → ?	C2 → ?	C2 → ?	

(c) Transition maps.

Figure 13: A-C example. Predicate summary: `P1(`α `: int)` :: (α`=0)`; `P2(`c `: C)` :: (c`.disposed=0)`; `P3(`c `: C)` :: (c`=null)`; `P4(`a `: A)` :: (a`.f=null)`; `P5(`a `: A)` :: (a`=null)`.

Sample	EA	CD	Slicing	AI
A-C	0.091	0.006	0.025	0.806
Set-Iterator	0.357	0.005	0.112	1.07
GRP	0.370	0.004	0.033	0.434
TooManyListeners	0.043	0.004	0.087	0.404
Queue	0.356	0.004	0.026	0.487
Stack	0.328	0.004	0.021	0.435
List Traversal	0.363	0.006	0.025	0.735

Table 5: Computation time in seconds

Sample	Number of Heap Objects
A-C	3 `A`, 4 `C`
Set-Iterator	5 `Set`, 6 `Iterator`, 6 `Version`
GRP	4 `Graph`, 4 `Iterator`
TooManyListeners	4 `DropTarget`, 4 `Listener`
Queue	7 `Element`, 3 `Queue`
Stack	7 `Element`, 3 `Stack`
List Traversal	7 `Element`, 3 `List`

Table 6: Size of heap space

java.awt that permits one and only one Listener to be registered on the particular event listener source concurrently. The `Stack` and `Queue` examples are standard implementations of data structures, and along with the `List`, were used to evaluate how our implementation handles recursive data structures.

Implementation Details. Our algorithm was built on top of a standard Java bytecode analyzer that takes Java bytecode and builds the control flow graph with all local variables in SSA form. Our algorithm consists of the following components - (*i*) Escape Analysis, (*ii*) Computing the control dependence, (*iii*) Slicing and Boolean Code Generation, and (*iv*) Abstract Interpretation and Client Emulation. Table 5 gives the compute times for each of the components for the sample programs described in this paper.

Table 6 shows the number of heap objects of each type that were generated for each of the examples. The number of objects ranges from 8 to 17. This shows that a relatively small heap space suffices to reason about these programs.

Although the running time is pretty reasonable for small examples, it is clear that scalability will be a concern for larger components. In future work, we plan to work with larger components and devise techniques for trading off precision with scalability.

7. RELATED WORK

Building on the early work on typestates [17], recent work has applied typestates to objects. In [8] a typestate system is presented that takes into account states of nested objects as well; however, unrestricted inter-object references are not permitted. In [11], a very general typestate system is presented in which typestate is expressed as membership in certain sets. This work can account for inter-object references in a fairly general way. However, in both these systems the typestate specification is entirely manual. The component writer must specify the preconditions and postconditions for each method. In contrast, our work attempts to create relevant typestates automatically based on a given reachability criterion.

A tool presented in [9] uses a built-in set of predicates to derive abstract object states, called roles, that are of interest to the programmer. In addition to being limited to the build-in predicates, the tool does not use static analysis; instead, it executes an instrumented version of the program and tracks the transitions that can be observed. The user is then expected to manually coarsen the abstraction, thus creating "role subspaces", in order to obtain usable problem-specific information. In contrast, the goal of this work is to automatically generate useful, problem-specific information using static analysis of a component.

Automatically determining specifications that describe *legal call sequences* for object-oriented components was pioneered by [18]. However, their work does not support aliasing and focuses on a single object. Their approach was also severely limited in the amount of *context* that could be expressed in the interface specification: the specification only described constraints based on the last method invoked on the component. The Jist [2] project improves on this technique by allowing the programmer to select the amount of context captured in the specification with a parameter k, resulting in a finite state machine with at most k states. Jist work does not account for the behavior of inter-object references. However, Jist takes certain steps to synthesize an interface that is easily understood. We might be able to leverage the Jist techniques to go from the heap state space to a readable interface specification.

An alternative approach [10] uses iterative, counter-example driven iterative refinement to create a safe and permissive but also possibly much larger interface. This approach separates the abstraction refinement for showing safety and permissiveness, which helps in obtaining parsimonious abstractions. While the described algorithm can be applied to components that use references, it is only guaranteed to terminate if the state of the component is finite.

The Canvas work [15] tries to create abstractions of provided components, using which client code can be checked for conformance. Canvas uses programmer supplied predicates to base predicate abstraction. Their focus is also on aliasing properties, and the Set-Iterator example in our work is inspired by their work. Canvas does not create object typestates–it is not its purpose. Rather, Canvas inlines a method's abstraction—essentially a boolean program—with abstracted client code and then uses TVLA [12] to verify correct usage in the client. By contrast, for limited variety of predicates, we construct method preconditions and postconditions without any client code being given.

The goal of shape analysis [12, 16] is finding precise heap abstractions that can model aliasing relationships between objects. The heap abstraction used by the three value logic analyzer (TVLA) system [12] uses a set of binary *abstraction predicates* to decide which objects should be summarized with only one node in the heap abstraction. For each of these nodes, TVLA then keeps 3-valued (true, false, unknown) information about various other predicates of interest. Similar to the work presented in this paper, TVLA uses abstract interpretation over a heap abstraction that is based on predicates to statically analyze programs. In contrast to our work, TVLA's abstract interpreter operates on an abstracted heap; our interpreter operates on concrete heaps and only uses the abstraction function to bound the exploration of the concrete synthetic heaps. Using concrete synthetic heaps for the interpretation has the

advantage that summary nodes can be avoided since all references are "must" pointers. However, using predicates on summary nodes TVLA is theoretically able express more general properties, in particular those that involve transitive closure. Since the predicate discovery used in this paper will not generate such predicates, this potential advantage is insignificant for the problem at hand.

Automatic discovery of predicates has previously done using counterexample-guided abstraction refinement [4, 6, 7]. In those analyses the algorithm checks if the counterexample that was found when trying to verify a property is spurious, and if so uses those spurious counterexamples to iteratively construct a refined abstraction. These analyses are typically limited to a fixed, finite abstract domain of cartesian products of boolean values. In the domain of shape-analysis [12, 16], recent work [13] described an algorithm that iteratively breaks the verification condition down into smaller abstraction predicates. In contrast, the work presented in this paper does not use complex programmer-provided verification conditions but uses reachability as the only input beyond the code itself.

8. CONCLUSION

We presented a new static analysis to derive relevant object typestates in the presence of inter-object references. Object typestates contain predicates on an object's local state as well as on the state of the objects it may refer to. This is important in order to be able to account for side-effects that inter-object references may cause. Our analysis works in three parts. It first uses reachability of a certain program point as the criteria for predicate discovery. It then performs predicate abstraction to convert each input method to a boolean program. Finally, it uses client emulation to explore heap states that help discover distinct typestates and transitions. This analysis enables further applications such as client code verification.

In future work, we plan to devise ways to infer an interface specification for each component from the heap state space. Such specification would then be used for statically checking client code for correct use of the component.

9. REFERENCES

[1] http://www.eclipse.org/, 2004.

[2] 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.

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

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

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, 2000.

[7] S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*. Springer-Verlag, November 2002.

[8] R. DeLine and M. Fahndrich. Typestates for objects, 2004.

[9] B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 313–324, New York, NY, USA, 2002. ACM Press.

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

[11] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *ACM SIGPLAN Notices*, 39(3):46–55, March 2004.

[12] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer-Verlag, 2000.

[13] A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement for 3-valued-logic analysis. Technical report, University of Wisconsin, Madison, WI, April 2004.

[14] M. G. Nanda and S. Ramesh. Pointer analysis of multithreaded Java programs. In *Proceedings of the ACM Symposium on Applied Computing*, 2003.

[15] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 83–94. ACM Press, 2002.

[16] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[17] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1), 1986.

[18] 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.