RC22145 (Log# W0108-015) (08/09/2001)
Computer Science/Mathematics

IBM Research Report

Deriving Specialized Heap Analyses for Verifying Component-Client Conformance

G. Ramalingam
IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York
rama@watson.ibm.com

Alex Warshavsky
IBM Haifa Research Laboratory and
School of Computer Science
Tel-Aviv University
walex@math.tau.ac.il

John Field
IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York
jfield@watson.ibm.com

Mooly Sagiv
School of Computer Science
Tel-Aviv University
sagiv@math.tau.ac.il

This page intentionally left blank.

Abstract

We are concerned with the problem of verifying ("certifying") whether the client of a software component *conforms* to the component's constraints for correct usage. We focus here on the *Concurrent Modification Problem* (CMP), a component-client conformance problem that can occur when certain classes defined by the Java Collections Framework are misused. Since CMP errors arise from relationships among a potentially unbounded number of objects, the problem cannot be defined as a simple finite-state property. Instead, we show that it is natural to specify it as a particular *must-alias* property for an *abstract heap* manipulated by the component. Computing this must-alias property would seem to be difficult, since the problem is contained in a class known to be intractable. However, we show that by exploiting the fact that the component's abstract heap is manipulated only through a well-defined interface, we can systematically *derive* efficient and accurate certification algorithms for CMP. For a restricted, but still quite useful class of client programs, we can perform precise *inter*procedural certification in polynomial time. We then generalize the approach to arbitrary client programs. Although this generalization yields some loss of precision in principle, the results produced by our prototype implementation of the certification algorithm in practice are quite accurate for the suite of benchmark programs studied. We believe the techniques presented here are likely to be applicable to other conformance problems as well.

1 Introduction

The CANVAS[1] project at IBM Research and Tel-Aviv University [8] aims to help ease the use of software components [53, 63] by

- Allowing the component designer to specify component *conformance constraints*, which describe correct component usage by a client program in a natural (yet still formal) way.
- Providing the client code developer with automated software *certification* tools to determine whether the client satisfies the component's conformance constraints.

For the purposes of this project, we consider a "component" to be any *object-oriented software library*, and focus on components written in Java [24].

1.1 The Concurrent Modification Problem

In this paper, we will extensively study a problem that is a microcosm of CANVAS's broader goals. The problem, which we will call the *Concurrent Modification Problem* (or CMP), arises in the context of the Java Collections Framework (JCF) [9]. JCF classes check for certain forms of inconsistent usage at runtime, and throw an exception when such usage occurs. One such runtime exception is the `ConcurrentModificationException` (or `CME`). Note that the name of the exception is rather misleading, since the problem can (and often does) manifest itself in programs with only a single thread.

Consider the Java fragment in Figure 1. The `Set` interface, the `Iterator` interface, and associated implementation classes are all part of JCF; classes implementing the `Iterator` interface are used to iterate over the contents of an underlying collection.

A fundamental requirement on the use of iterators is that once an `Iterator` object o_i is created on a collection o_c, it may be used only as long as the collection o_c is not modified, *not counting modifications made via* o_i. This ensures that the internal state invariants of o_i are not corrupted by another iterator, or by direct update to the collection. JCF collections detect violations of this constraint *dynamically*, and raise `CME` when it occurs. In the programs we have studied, we have found that `CME` is often a manifestion of subtle logical errors. Hence, statically determining if a program may throw `CME` can often help to identify the presence of *related* errors at compile time, even in the absence of a formal specification of the client program's behavior.

In the code fragment of Figure 1, an iterator is created on a `worklist`, which is implemented using a JCF `HashSet`. The iterator is then used to process each item on the worklist in succession. We observe that `CME` can be raised during item processing, since the nested call to `doSubproblem(...)` causes `worklist.addItem(newitem)` to be called, which will in turn attempt to update the underlying `HashSet` while the iterator is still active. On the next iteration, the call to `i.next()` would cause `CME` to be thrown. This code fragment is actually an abstraction of situations we encountered in the Soot [64] framework and the TVLA [46] system (both used to implement the algorithms we present in the sequel) where `CME` was raised. We note that CMP errors are apparently quite widespread in practice [54, 4, 37, 5, 40].

1.2 Our Approach to Conformance Certification

In the case of CMP, certain types of client misuse of a component are detected dynamically. The goal of our work is to detect all possible instances of client non-conformance statically. Using CMP as a running example, the remainder of this paper addresses the component conformance certification problem by pursuing the following themes:

[1] *C*omponent *AN*notation, *V*erification, *A*nd *S*tuff

```
class Make {
   private Worklist worklist;
   public static void main (String[] args) {
      Make m = new Make();
      m.initializeWorklist(args);
      m.processWorklist();
   }
   void initializeWorklist(String[] args) {
      ...; worklist = new Worklist(); ...
      // add some items to worklist
   }
   void processWorklist() {
     Set s = worklist.unprocessedItems();
     for (Iterator i = s.iterator(); i.hasNext()){
       Object item = i.next(); // CME may occur here
       if (...) processItem(item);
     }
   }
   void processItem(Object i) {
      ...; doSubproblem(...);
   }
   void doSubproblem(...) {
      ... worklist.addItem(newitem); ...
   }
}
public class Worklist {
   Set s;
   public Worklist() {
      ...;  s = new HashSet(); ...
   }
   public void addItem(Object item) { s.add(item); }
   public Set unprocessedItems() { return s; }
}
```

Figure 1: An erroneous Java program fragment throwing `CME`.

Program specification via heap properties

Heap or pointer analysis is frequently regarded as an irritating but unavoidable auxiliary step that must be carried out prior to analyzing some program property of immediate interest. By contrast, we show in this paper that it can be quite natural to *express* a component conformance constraint directly as some heap property. In Section 2.2, we show in detail how this can be done for CMP.

Our approach makes it possible to model richer forms of behavior than the finite state or temporal properties addressed by many software verification systems [20, 26, 48]. In particular, heap properties can be used to model the interactions among a potentially *unbounded* number of component objects, a prerequisite for analyzing CMP.

In the sequel, we will use *abstract Java programs* (written in an experimental specification language called EASL[2]) for specifying the conformance constraints of the component that clients must satisfy. While we believe that the use of such an "executable" specification language has certain advantages, it is not critical to the results presented here. Indeed, more traditional declarative specification languages such as JML [43] can also express many of the properties with which we are concerned.

Off-line generation of problem-specific heap analyses

Having made the decision to express a component's conformance constraints in the form of heap properties, we could attempt to verify such properties using standard algorithms for alias, pointer, or shape analysis (e.g., [38, 51, 27, 61, 60]). We investigated the applicability of several heap analysis algorithms to CMP, including a powerful *relational* [52, p. 248] heap analysis implemented using the TVLA system [46]. However, it soon became apparent that even the most powerful generic heap analysis was insufficiently precise for CMP.

Therefore, instead of using generic heap analysis techniques, we exploit the following properties of the conformance constraint certification problem:

- The parts of the (unbounded) heap relevant to the component are manipulated only through a *well-defined interface*.
- The heap properties of interest are limited to those necessary to carry out client certification.

We will thereby show how an *off-line analysis* of the component's conformance constraint can be used to systematically derive a *problem-specific* heap analysis that is suitable for *any* client that uses the component. Not only

[2] *E*xecutable *A*bstraction/*S*pecification *L*anguage

does this approach yield a more precise analysis than the generic approach, we have found that it also yields one with much better performance in practice.

Our technique for off-line analysis generation is based on the idea of attempting to compute for each conformance constraint φ the *weakest precondition* [17] for φ to hold under all potential sequences of component/client interactions. We use these derived properties to systematically derive an *instrumented* [52] semantics for any client program, where the instrumentation maintains properties relevant to the certification problem at hand. The instrumented semantics can in turn be used as the basis for generating problem-specific client analyzers.

1.3 Results

Our principal results are as follows:

- We provide a straightforward and intuitive formal specification of CMP as a heap property.
- We show how to use an iterated weakeast precondition computation to *systematically* derive a set of static analysis algorithms for CMP.
- For Java programs in which references to collections and iterators are stored only in local (stack) or static variables, (rather than in object fields), we provide precise, polynomial-time intra- and *inter*-procedural certification algorithm for detecting instances of CMP. We will refer to this restricted verison of CMP as SCMP ("shallow" CMP). This result may be somewhat surprising, since SCMP, expressed in heap analysis terms, is a must-alias problem with 3-level pointers—a class for which must-alias analysis is in general *PSPACE*-hard in the intraprocedural case and *EXPTIME*-hard in the interprocedural case [50]. Note that many existing JCF applications are likely to be members of the SCMP class, since the SCMP class does include programs that allocate an unbounded number of collections or iterators.
- SCMP focuses on analyzing the heap that is *internal* to `Set` and `Iterator` components, but does not address the question of how to analyze the *client*'s heap, which may contain objects containing references to collections and iterators. Since we want to formulate an analysis without access to the client code (or a specification of the client code), we must of necessity use a generic heap analysis for handling the client's heap. We show how to handle the unrestricted CMP problem by exploiting the same instrumentation generation approach used for SCMP. The extended algorithm is also *adaptive*, in the sense that it yields precise results in polynomial time when the client program is in the SCMP class.
- We measure the precision and running times of a prototype implementation of our algorithm for the unrestricted CMP problem on a number of benchmarks. The benchmarks include both contrived programs intended to test the algorithm's strengths and weaknesses, and larger "real-world" programs that use JCF. The algorithm produces minimal "false alarms" on these benchmarks, with reasonable speed.
- Our technique for deriving precise analyses can be formally viewed as a process for generating a problem-specific instrumented semantics for the client program. In particular, the technique can be used to derive *instrumentation predicates* for the TVLA system. In previous analysis algorithms developed using TVLA [45, 69], the instrumentation predicates, which are critical to the performance and precision of the analysis, were developed in an *ad hoc* manner.

2 CANVAS and the Concurrent Modification Problem

The CANVAS component certification approach separates the component conformance constraint certification problem into several phases:

1. The component designer or implementer writes a conformance constraint specification. Such a specification will typically describe properties of the component that the designer regards as critical to correct client usage. However, the specification need not (and usually should not) specify all of the component's correct or incorrect behaviors. We assume that such specifications will generally be written manually, in part because a certain amount of judgement is required to determine those aspects of the component's behavior that are most pertinent to correct usage. However, extracting candidate constraint specifications from a concrete implementation is an interesting alternative approach.
2. The component's conformance constraint specification is "compiled" and analyzed to produce a component-specific conformance constraint certification program (or *certifier*). The certifier will typically be generated once (or at least infrequently), but used many times. Hence sophisticated, expensive, or even manual analysis techniques can be exploited at "certifier generation time" to yield a precise and efficient certifier.
3. The client program[3] is fed to the certifier, and a report is produced indicating whether possible viola-

[3] In the sequel, we will assume there is only a single component

tions of the conformance constraints have occurred.

In this paper, the somewhat idealized process outlined above is realized as follows:

- Phase 1 uses a specification language called EASL/P, described in more detail in Section 2.1.
- In Phase 2, the EASL/P specification is translated to the *TVP* language (see Section 3) used as input to the TVLA [46] program analysis system. In Section 3, an EASL/P specification of each method of the component is translated in a natural way to TVP. In Section 4, a more sophisticated analysis of the specification generates an instrumented TVP input that results in a much more precise and efficient analysis.
- In Phase 3, the Soot Java bytecode analysis framework [64] is used to generate an intermediate representation of the client program. This representation is then translated in a straightforward manner to TVP, optionally guided by the instrumented semantics determined in Phase 2. Finally, the TVP specification of each component method generated in Phase 2 is "linked" with the client specification by inlining each call site to a component method in the client TVP code with the component specification's TVP code. Details of this phase are covered in Section 5.

2.1 The Specification Language EASL/P

CANVAS's EASL specifications take the form of *abstract* Java programs. In this paper, we will focus on a subset of EASL we call EASL/P. The syntax of EASL/P is depicted in Figure 2. EASL/P combines a restricted subset of Java statements (assignments, conditionals, loops, and heap allocation) and a restricted set of primitive types, namely booleans and pointers, with a *requires* statement. These constructs are sufficient to simulate the more conventional pre- and post-condition specification style (as exemplified, e.g., by JML [43]), as well as various forms of finite state specifications. Most importantly, EASL/P allows a natural expression of component behaviors that determine the relationships among potentially unbounded numbers of component objects.

While a discussion of the merits of an "imperative" specification language is beyond the scope of this paper, we note that EASL is similar in spirit to several existing "wide spectrum languages" that combine specification statements with conventional programming constructs [49, 31]. In any case, for the applications discussed here the choice of specification language is immaterial, provided that the language chosen has expressive power equivalent to EASL/P.

and a single client; however, our approach is intended to accommodate clients that use multiple components, or components that incorporate other components.

```
⟨Component⟩  →  class ⟨Id⟩ { ⟨Field⟩* ⟨Method⟩* }
⟨Field⟩      →  ⟨Type⟩ ⟨Id⟩;
⟨Type⟩       →  Boolean | ⟨Id⟩
⟨Method⟩     →  ⟨Type⟩ ⟨Id⟩ (⟨Arg⟩⊛) { ⟨Stmt⟩* }
⟨Arg⟩        →  ⟨Type⟩ ⟨Id⟩
⟨Stmt⟩       →  requires ⟨Cond⟩;
             |  ⟨Lhs⟩ := ⟨Exp⟩;
             |  { ⟨Stmt⟩* }
             |  while ⟨Cond⟩ do ⟨Stmt⟩;
             |  if ⟨Cond⟩ ⟨Stmt⟩ [ else ⟨Stmt⟩ ] ;
             |  return ⟨Exp⟩;
⟨Lhs⟩        →  this | ⟨Id⟩ | ⟨Lhs⟩.⟨Id⟩
⟨Exp⟩        →  null | new ⟨Id⟩ (Exp⊛)
             |  ⟨Lhs⟩ | ⟨Cond⟩
⟨Cond⟩       →  true | false | ⟨Lhs⟩
             |  ⟨Exp⟩ == ⟨Exp⟩
             |  !⟨Cond⟩
             |  ⟨Cond⟩ and ⟨Cond⟩
             |  ⟨Cond⟩ or ⟨Cond⟩
```

Figure 2: The EASL/P language. X^* denotes zero or more occurrences of X, while $X^{\circledast}$ denotes a comma-separated list of zero or more occurrences of X.

2.2 Specifying CMP

We now show how CMP can be specified in EASL/P by abstracting the concrete implementation used to detect CMP at runtime into a heap property.

The standard Java collection classes detect CME at runtime by maintaining a counter, `modCount`, which holds the number of times that a given collection has been structurally modified. The `modCount` of a collection is recorded by an iterator object at the time an iterator on the collection is created, and whenever the iterator updates the underlying collection. When the `next`, `remove`, or similar methods are invoked on the iterator, the `CME` exception is thrown if the underlying collection's current `modCount` value differs from the value associated with the iterator's creation[4].

[4]Strictly speaking, the operations on `modCount` differ slightly from the description here to avoid the need for synchronized (monitor-mediated) access; these differences are irrelevant in the context of this paper

Figure 3 contains an EASL/P specification of CMP. The specification is an abstraction of the actual implementation. Rather than use an integer counter `modCount`, we use dynamically-allocated instances of the class `Version` to distinguish the "version" of a collection induced by the last update. As we shall see, the simple form of the specification will make it possible to derive an efficient static analysis without the need to reason about integer arithmetic. Note, however, that while the specification of Figure 3 does not require reasoning about integers, it does require reasoning about a potentially unbounded collection of version objects. In particular, it requires computing certain must-alias facts.

2.3 A Running Example

We will use the Java fragment in Figure 4 as a running example to illustate our algorithm. In this example, the collection `v` is modified via iterator `i1` in line `5`, which renders iterator `i2`, over the same collection `v`, *invalid*. If iterator `i2` is used in the subsequent line `6`, the method call `i2.next()` will cause the `CME` exception to be thrown. However, the use of iterator `i3` on line `7` is still valid, since both `i1` and `i3` refer to the same iterator. The direct update of `v` on line `8` invalidates *all* iterators over `v`, which will cause `CME` to be thrown on line `9`, if `i1.next()` is executed.

This example illustrates why it is difficult to ensure that an analysis of CMP is both sound and avoids excessive false alarms. Note that at line `6`, CME is thrown because `i2` and `i1` refer to *different* iterators over the *same* set. An analysis that misses this error is unsound. In contrast, CME is not thrown at line `7`, since `i3` and `i1` refer to *same* iterator.

3 Applying Standard Heap Analyses to CMP

In this section, we show how, given an EASL/P specification of a component, we can certify clients of the component using standard "off the shelf" heap analyses, using CMP to illustrate the approach. Unfortunately, the certification algorithms obtained by using standard heap analyses are either imprecise or expensive or both even for the simpler SCMP problem. We begin by describing TVP, a language based on first order logic for specifying operational semantics, and TVLA, a system for generating abstract interpreters from a TVP semantic specification [46]. We present a simple TVP semantics for EASL/P. This semantics, in conjunction with TVLA, yields an algorithm for CMP that we will refer to as VCMP (for "vanilla" CMP). Readers not interested in the formal details may skip to Section 3.3.

3.1 A TVP Semantics for EASL/P

Program states are represented in TVP by *2-valued logical structures.* A 2-valued logical structure $S^\natural$ over a set of predicates P is a pair $\langle U^\natural, \iota^\natural \rangle$ where:

- $U^\natural$ is a set referred to as the universe of $S^\natural$, and
- $\iota^\natural$ is an interpretation function that maps predicates to their truth-values in the structure, i.e., for every predicate $p \in P$ of arity k, $\iota^\natural(p)\colon {U^\natural}^k \to \{0,1\}$.

We model the program state of an arbitrary EASL/P program using a 2-valued structure as follows:

- The universe of the structure represents the set of (heap allocated) objects.
- Every reference variable `var` is represented by a unary predicate $pt[\texttt{var}]$; the value of $pt[\texttt{var}](o)$ is true iff `var` refers to ("points to") the object o. (In the sequel, we will frequently use the notation $p[\cdot]$ to denote a family of predicates indexed by some construct derived from the program).
- Every class field `fld` of a reference type is represented by a binary predicate $rv[\texttt{fld}]$ ("rvalue"); the value of $rv[\texttt{fld}](o_1, o_2)$ is true iff the field `fld` of object o_1 refers to object o_2.

Note that there may be no bound on the size of the universe required to model the states produced by a given program, as the program may allocate an unbounded number of objects.

In this paper we make use of only 2-valued structures consisting of nullary, unary, and binary predicates. We will depict such 2-valued states as directed graphs as follows: Each individual from the universe is displayed as an ellipse. A unary predicate p is represented by a rectangle labelled p. An edge is drawn from the box representing p to the ellipse representing an individual u iff $p(u)$ is true. For every binary predicate q, an edge labelled q is drawn from an ellipse representing u_1 to an ellipse representing u_2 iff $q(u_1, u_2)$ is true. The name inside an ellipse is used only for reference purposes, it is not not used to define the structure.

Figure 5 depicts a 2-valued structure representing the state of the program of Figure 4 after the execution of the statement 5. The state precisely indicates that $i1$ and $i3$ point to the same iterator object, which is valid, while $i2$ points to a different iterator, which is invalid.

The semantics of a program statement is specified in TVP using *actions*, which represent transformers of 2-valued structures. Actions are specified using first order logical formulae over the underlying set of predicates (and may include universal and existential quantifiers as well

```
class Version { /* class representing distinct "versions" of the same collection */ }
class Set {
   Version ver;
   Set() { ver = new Version(); }
   boolean add(Object o) { ver = new Version(); }
   Iterator iterator() { return new Iterator(this); }
}
class Iterator {
   Set set;  // set on which iterator is defined
   Version defVer; // set version on which iterator was created
   Iterator (Set s){ defVer = s.ver; set = s; }
   void remove() {
      requires  (defVer == set.ver);
      set.ver = new Version();
      defVer = set.ver;
   }
   Object next() {
     requires (defVer == set.ver);
   }
}
```

Figure 3: An EASL/P specification of the concurrent modification problem.

```
/* 0 */    Set v = new Set(); // create a new set
/* 1 */    Iterator i1 = v.iterator();
/* 2 */    Iterator i2 = v.iterator();
/* 3 */    Iterator i3 = i1;
/* 4 */    i1.next(); // iterate to the next element
/* 5 */    i1.remove(); // update v via i1; different iterators on v invalidated
/* 6 */    if (...) i2.next(); // CME may be thrown
/* 7 */    if (...) i3.next(); // i3 still valid; CME will not be thrown
/* 8 */    v.add("a new string"); // all iterators over v invalidated
/* 9 */    if (...) i1.next(); // CME may be thrown
```

Figure 4: A Java fragment illustrating CMP

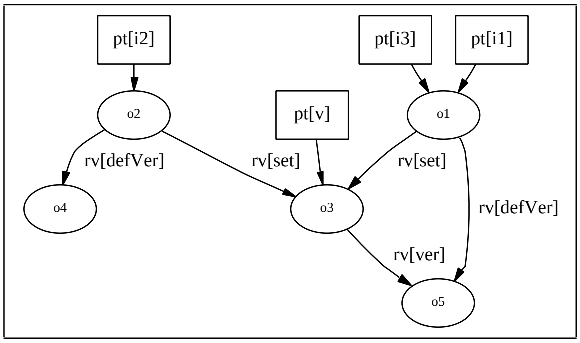

Figure 5: A 2-valued structure representing the concrete program state at label 6.

EASL/P Condition	TVP Precondition
`x == y`	$\forall o : pt[\mathtt{x}](o) \Leftrightarrow pt[\mathtt{y}](o)$

EASL/P Statement	TVP Semantics
`x = new C()`	let $n = \text{new}()$ in $pt[\mathtt{x}](o) := (o = n)$
`x = y`	$pt[\mathtt{x}](o) := pt[\mathtt{y}](o)$
`x = y.fld`	$pt[\mathtt{x}](o) := \exists o_1 : pt[\mathtt{y}](o_1) \wedge rv[\mathtt{fld}](o_1, o)$
`x.fld = y`	$pt[\mathtt{x}](o_1) \triangleright rv[\mathtt{fld}](o_1, o_2) := pt[\mathtt{y}](o_2)$

Table 1: A TVP semantics for the pointer manipulation statements of EASL/P.

as the equality predicate with the standard interpretation.) Specifically, an action comprises of the following parts:

- An optional *precondition* formula: actions without a precondition are unconditional and apply to all structures; an action with a precondition is a conditional one, which applies only to structures satisfying the precondition.
- Zero or more *allocation* bindings of the form "`let id = new() in`", which adds a new element to the universe, which may be referred to by `id` in the following update; the value of any predicate $p(v_1, \ldots, v_k)$, where at least one of the v_i represents a newly added element, is defined to be false.
- Zero or more predicate definitions (also referred to as update formulate) of the form $p(v_1, v_2, \ldots, v_m) := \varphi(v_1, v_2, \ldots, v_m, n_1, \ldots, n_k)$ where each n_i is a variable bound by an allocation binding, which defines the value of predicate p using a formula.

We will use the notation $guard \triangleright lhs := rhs$ as shorthand for the following predicate definition:

$$lhs := (guard \wedge rhs) \vee (\neg guard \wedge lhs)$$

A TVP program is a (control-flow) graph, each edge annotated with an action.

Table 1 presents a straightforward TVP semantics for typical pointer manipulation statements. (Note that the semantics specified for `x = new C()` models just the allocation of a new object. Initialization of the object by the constructor is modelled separately.)

Table 2 depicts the TVP semantics assigned to calls in the client code to the methods of the `Set` and `Iterator` class. The semantics is obtained by applying the semantics shown in Table 1 to the method specifications shown in Figure 3, and slightly simplifying the results. The methods `i.next()` and `i.remove()`, which have a `requires` clause, are modelled by a pair of parallel edges, one (the "true" branch) representing the case when the `requires` clause is met and the other (the "false" branch) representing the case when it is not. The `requires` condition is translated into a corresponding TVP precondition on these edges. Table 2 shows the TVP semantics associated with the "true" branch only.

Consider the semantics of the statement `v = new Set()`. We first allocate two individuals n_1, representing the new `Set` object, and n_2, representing the new `Version` object created by the `Set` constructor. Then, we model the assignment to `v` and the *ver* field of the newly created `Set` object just as we modelled reference assignments in Table 1.

As another example, consider the statement `i.remove()`. This creates a new `Version` object. The guard $pt[\mathtt{i}](o)$ identifies the iterator object o referred to by variable `i`), and the defining version of o is updated (to refer to the newly created version) by updating predicate $rv[\mathtt{defVer}]$. The formula $\exists o : pt[\mathtt{i}](o) \wedge rv[\mathtt{set}](o, o_2)$ identifies the `Set` o_2 for which `i` is an iterator, and updates the version associated with the collection by updating the value of the predicate $rv[\mathtt{ver}](o_2, o_1)$.

3.2 Generating An Abstract Interpreter From the TVP Semantics

The TVLA system generates abstract interpreters from TVP semantic specifications, using abstract domains based on 3-valued logic. Three-valued logic extends boolean logic by introducing a third value 1/2 denoting values which may be 1 or 0.

2-valued structures are just special cases of 3-valued structures. 2-valued structures (and even 3-valued structures) can be *abstracted* into smaller 3-valued structures by merging multiple individuals into one, and by approx-

Method Call	Method's Precondition in TVP
`i.next()` `i.remove()`	$\exists o, s : pt[\texttt{i}](o) \wedge rv[\texttt{set}](o, s) \wedge$ $\forall v : rv[\texttt{defVer}](o, v) \Leftrightarrow rv[\texttt{ver}](s, v)$

Method Call	TVP Semantics
`v = new Set()`	let $n_1 = \text{new}()$ in let $n_2 = \text{new}()$ in $pt[\texttt{v}](o_1) := (o_1 = n_1)$ $(o_1 = n_1) \triangleright rv[\texttt{ver}](o_1, o_2) := (o_2 = n_2)$
`v.add()`	let $n_1 = \text{new}()$ in $pt[\texttt{v}](o_2) \triangleright rv[\texttt{ver}](o_2, o_1) := (o_1 = n_1)$
`i = v.iterator()`	let $n_1 = \text{new}()$ in $pt[\texttt{i}](o_1) := (o_1 = n_1)$ $(o_1 = n_1) \triangleright rv[\texttt{set}](o_1, o_2) := pt[\texttt{v}](o_2)$ $(o_1 = n_1) \triangleright rv[\texttt{defVer}](o_1, o_2) := \exists o : pt[\texttt{v}](o) \wedge rv[\texttt{ver}](o, o_2)$
`i.remove()`	let $n_1 = \text{new}()$ in $pt[\texttt{i}](o) \triangleright rv[\texttt{defVer}](o, o_1) := (o_1 = n_1)$ $\exists o : pt[\texttt{i}](o) \wedge rv[\texttt{set}](o, o_2) \triangleright rv[\texttt{ver}](o_2, o_1) := (o_1 = n_1)$

Table 2: A TVP semantics for calls to `Set` and `Iterator` methods.

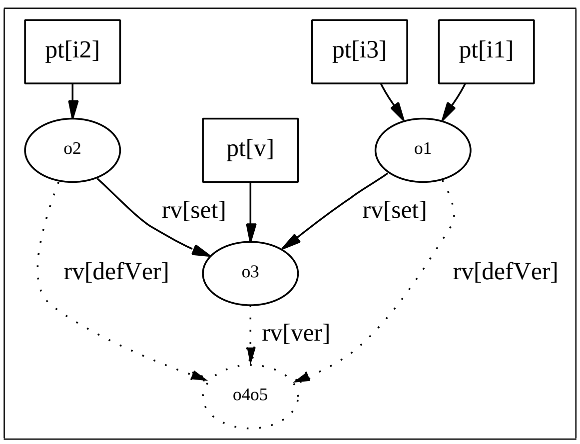

Figure 6: A 3-valued structure (abstract state) that approximates the 2-valued structure (concrete state) shown in Figure 5.

imating the predicate values appropriately. Figure 6 illustrates this abstraction process. The figure depicts a 3-valued structure obtained from the 2-valued structure shown in Figure 5 by merging nodes $o4$ and $o5$. Dotted edges indicate predicate values that are $1/2$, while dotted nodes indicate individuals obtained by merging multiple individuals.

One way of identifying individuals that may be merged is to use *canonical abstraction*. Let $A \subseteq P$ be a set of unary predicates, which we refer to as *abstraction predicates*. Canonical abstraction merges all individuals having the same value for all abstraction predicates into one abstract individual. We will refer to an abstract individual representing multiple concrete individuals as a *summary node*. Our "vanilla" algorithm for CMP, VCMP, is obtained by treating all unary predicates as abstraction predicates (TVLA's default behavior): *i.e.*, every predicate of the form $pt[\texttt{var}]$ is an abstraction predicate.

The maximum number of nodes in a structure produce by canonical abstraction is $3^{|A|}$. Hence, the number of different abstract structures that can arise at a program point is bounded and can be computed. TVLA implements a standard iterative algorithm to compute the set of all abstract structures that can arise at every program point (a relational analysis). TVLA also implements a corresponding independent attribute analysis that computes a single structure at every program point that approximates all structures that may arise at that point.

In VCMP, $|A|$ is the number of reference variables. Consequently, the complexity of VCMP is doubly exponential in the number of reference variables in the worst case.

3.3 The Effectiveness of Standard Heap Analyses

VCMP, the certifier produced by the simple semantics described above, incorporates a heap analysis that merges nodes in a structure together iff they are pointed to by the same set of variables [66, 59]. It is straightforward [60] to use various other alternative semantics to generate alternative heap analyses, such as [39, 10] or [38]. By simply choosing between various of these semantics, we produce certifiers for CMP that utilize a corresponding must-alias analysis algorithm.

Let us see what happens when we apply VCMP to our running example. The abstract state in Figure 6 *represents* the concrete program state shown in Figure 5. The two concrete nodes for the two version objects $o4$ and $o5$ are blurred together because there is no abstraction predicate in the concrete structure that differentiates between them. As a result of *merging* together $o4$ and $o5$, we have lost information: in the abstract state, we will have to conservatively assume that each of `i1` `i2`, and `i3` may be *either valid or invalid*, while in the concrete state it is clear that `i1` and `i3` are *valid*.

While some heap analyses that use allocation-site based merging will produce the right result in this example, it is easy to construct other examples (based on loops containing allocation sites) for which allocation-site based merging will produce imprecise results. Any relational analysis based on k-limiting, for $k \geq 3$ is precise, but doubly exponential, for SCMP. Further, such k-limiting based algorithms can become very imprecise in the general case, when collections may be embedded deep inside the client's data structures.

4 Systematic Generation of CMP-Specific Heap Analysis

Note that a component's heap can be manipulated (by the client) only through a restricted and well-defined interface. Consequently, the specification of the component's interface can be used to derive a specialized heap analysis (for the component's heap), which can be used as the basis for more effective certification algorithms. In this section, we illustrate this by presenting a precise, polynomial time algorithm for SCMP, derived from the EASL/P specification using a systematic procedure. The derivation procedure can be applied to arbitrary EASL/P specifications, but may not terminate in the general case. While heuristics can be used to ensure termination, devising termination strategies that generate analyses that are both sufficiently precise *and* efficient from arbitrary EASL/P specifications is an interesting open problem.

4.1 A Specialized Analysis For SCMP

The essential steps in deriving a specialized analysis are (a) determining the information about a component's state that is relevant to the certification process (referred to as *instrumentation predicates*[5]) and (b) determining how this relevant state information is changed by any call to a component method. This information yields a specialized semantics for the component's method that serves as the basis for the analysis.

We identify instrumentation predicates using a backward, symbolic analysis, based on weakest-precondition computation [17], over any possible sequence of component method calls by using the following rules iteratively:

1. If any component method has a "`requires` φ" clause at method entry, then $\neg\varphi$ is a *candidate instrumentation formula.*
2. If $\varphi_1 \vee \ldots \vee \varphi_k$ is a candidate instrumentation formula, then each φ_i is a *candidate instrumentation predicate.*
3. If φ is a candidate instrumentation predicate, and `S` is the body of a component method, then the weakest precondition of φ with respect to `S`, $WP(\mathtt{S}, \varphi)$, is a *candidate instrumentation formula.* ($WP(\mathtt{S}, \varphi)$ is a formula that holds before the execution of `S` iff φ holds after the execution of `S`.)

The above rules ignore "`requires`" clauses at points other than at method entry (the CMP specification does not use any such clause). Such clauses, however, may be handled using similar iterative weakest-precondition computation over the *statements* of the method's specification.

We now illustrate this procedure for SCMP. For the sake of readability, we will express formulae using EASL/P notation rather than first order logic.

Step 1: In SCMP, we are interested in determining at every call-site to methods `Iterator::next()` and `Iterator::remove()`, say, on an `Iterator` variable `i`, if the precondition of the methods may fail, that is, if `i.defVer != i.set.defVer` may be true. VCMP is imprecise because the information it maintains does not enable it to precisely test for this condition once abstraction has occurred. We therefore introduce a new predicate *invalid*[`i`] to represent the formula `i.defVer != i.set.defVer`.

Step 2: The next step is to consider how the execution of different `Set` and `Iterator` methods will affect the value of the predicate *invalid*[`i`]. As an example, consider the execution of the method call `v.add()`, where `v` is of type `Set`. It can be verified that *invalid*[*i*] is true after the execution of `v.add()` iff the condition (*invalid*[`i`] or (`i.set == v`)) is true before the execution of the statement. This suggests maintaining the value of the expression `i.set == v` in order to precisely update the value

[5] Instrumentation predicates [60]), are defined by formulae over the original, "core" state definition predicates. An instrumentation predicate adds no new information to a 2-valued structure, as its value can be computed from the values of core predicates using the defining formula. However, maintaining the value of an instrumentation predicate can make a 3-valued structure more precise, as the maintained value may be more precise than the value obtained by evaluating the defining formula in a structure containing predicates with value 1/2.

Predicate	Meaning (in EASL/P)	Defining formula (in TVP)
$invalid[\mathtt{i}]$	`i.defVer != i.set.defVer`	$\exists\ o, s, v : pt[\mathtt{i}](o) \wedge rv[\mathtt{set}](o, s) \wedge rv[\mathtt{defVer}](o, v) \not\Leftrightarrow rv[\mathtt{ver}](s, v)$
$iteratesOver[\mathtt{i}, \mathtt{v}]$	`i.set == v`	$\exists o_1, o_2 : pt[\mathtt{i}](o_1) \wedge pt[\mathtt{v}](o_2) \wedge rv[\mathtt{set}](o_1, o_2)$
$mutex[\mathtt{i}, \mathtt{j}]$	`(i.set == j.set) and (i!= j)`	$\exists\ o_1, o_2, o : pt[\mathtt{i}](o_1) \wedge pt[\mathtt{j}](o_2) \wedge o_1 \neq o_2 \wedge rv[\mathtt{set}](o_1, o) \wedge rv[\mathtt{set}](o_2, o)$
$sameSet[\mathtt{v},\mathtt{w}]$	`v == w`	$\exists s : pt[\mathtt{v}](s) \wedge pt[\mathtt{w}](s)$

Table 3: The nullary instrumentation predicates used by SCMP.

of $invalid[\mathtt{i}]$. Hence, we introduce a second instrumentation predicate $iteratesOver[\mathtt{i}, \mathtt{v}]$, representing the condition `i.set == v`.

Step 3: Similarly, consider the effect of executing the statement `j.remove()` (where `j` is an iterator variable) on the predicate $invalid[\mathtt{i}]$. It can be verified that $invalid[\mathtt{i}]$ is true after the execution of `j.remove()` iff the condition ($invalid[\mathtt{i}]$ or (`(i.set == j.set) and (i != j)`)) is true before the execution of the statement. We introduce the instrumentation predicate $mutex[\mathtt{i}, \mathtt{j}]$, representing the condition `(i.set == j.set) and (i != j)`.

Step 4: It can be verified that $iteratesOver[i, v]$ is true after the execution of `i = w.iterator()` iff `v == w` before the execution of the statement. We introduce the instrumentation predicate $sameSet[\mathtt{v}, \mathtt{w}]$, representing the condition `v == w`.

Interestingly, this procedure has reached a fixed point with respect to the SCMP problem. Given any input instance Pgm of SCMP, let IV and VV denote respectively the set of `Iterator` variables and the set of `Set` variables in the program. Define the set of instrumentation predicates to be

$$\begin{aligned} IP \;=\; & \{\ invalid[\mathtt{i}] \mid \mathtt{i} \in IV\ \} \cup \\ & \{\ iteratesOver[\mathtt{i}, \mathtt{v}] \mid \mathtt{i} \in IV, \mathtt{v} \in VV\ \} \cup \\ & \{\ mutex[\mathtt{i}, \mathtt{j}] \mid \mathtt{i}, \mathtt{j} \in IV\ \} \cup \\ & \{\ sameSet[\mathtt{v}, \mathtt{w}] \mid \mathtt{v}, \mathtt{w} \in VV\ \}. \end{aligned}$$

The value of any instrumentation predicate in IP after the execution of any statement in Pgm can be expressed in terms of the value of these instrumentation predicates before the execution of the statement. In other words, not only do we no longer require additional instrumentation predicates, we do not even need the core predicates in order to update the value of the instrumentation predicates!

Table 3 presents a formal definition of the instrumentation predicates using core predicates and explains their intended meaning in EASL/P. The update formulae for the instrumentation predicates is shown in Table 4. As the table shows, copy assignments of `Set` or `Iterator` variables have an obvious corresponding semantics. We refer to the independent attribute analysis generated by this specialized semantics as the SCMP analysis algorithm.

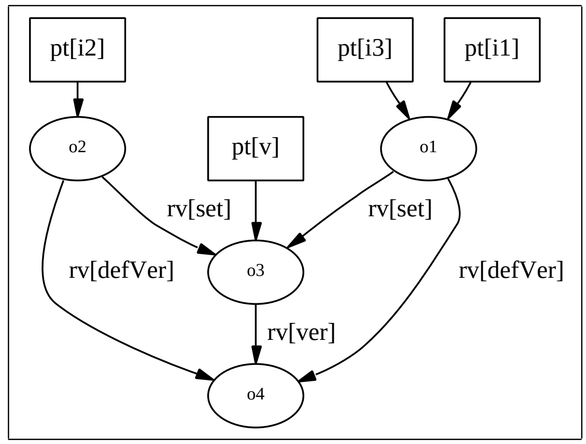

Figure 8: An illustration of the abstract state at program point `5` (*i.e.*, just before the execution of the statement labelled `5`), as computed by VCMP.

Figures 8 and 6 illustrate the abstract state maintained by the VCMP algorithm described earlier just before and after the statement `i1.remove()` in the program of Figure 4, leading to imprecision. Figure 7 illustrates the abstract state maintained by the more precise SCMP algorithm at these program points: the state representation is much more compact, yet more precise.

4.2 The Complexity of the SCMP Algorithm

The specialized semantics in Table 4 yields (the "dataflow equations" of) a distributive dataflow analysis problem over V^2 "bits", where V denotes the number of iterator and collection variables in the program. The problem is,

EASL/P **Statement**	**Specialized TVP Semantics**
`v = new Set()`	$sameSet[\mathtt{v},\mathtt{v}] := 1$ $sameSet[\mathtt{v},z] := 0$ for $z \in VV - \{\mathtt{v}\}$ $sameSet[z,\mathtt{v}] := 0$ for $z \in VV - \{\mathtt{v}\}$ $iteratesOver[z,\mathtt{v}] := 0$ for $z \in IV$
`v.add()`	$invalid[z] := invalid[z] \vee iteratesOver[z,\mathtt{v}]$ for $z \in IV$
`i = v.iterator()`	$iteratesOver[\mathtt{i},z] := sameSet[\mathtt{v},z] for z \in VV$ $mutex[\mathtt{i},z] := iteratesOver[z,\mathtt{v}]$ for $z \in IV - \{\mathtt{i}\}$ $mutex[z,\mathtt{i}] := iteratesOver[z,\mathtt{v}]$ for $z \in IV - \{\mathtt{i}\}$ $invalid[\mathtt{i}] := 0$
`i.remove()`	$invalid[\mathtt{j}] := invalid[\mathtt{j}] \vee mutex[\mathtt{j},\mathtt{i}]$ for $\mathtt{j} \in IV$
`v = w`	$sameSet[\mathtt{v},z] := sameSet[\mathtt{w},z]$ for $z \in VV - \{\mathtt{v}\}$ $sameSet[z,\mathtt{v}] := sameSet[z,\mathtt{w}]$ $z \in VV - \{\mathtt{v}\}$ $iteratesOver[z,\mathtt{v}] := iteratesOver[z,\mathtt{w}]$ for $z \in IV$
`i = j`	$iteratesOver[\mathtt{i},z] := iteratesOver[\mathtt{j},z]$ for $z \in VV$ $mutex[\mathtt{i},z] := mutex[\mathtt{j},z]$ for $z \in IV - \{\mathtt{i}\}$ $mutex[z,\mathtt{i}] := mutex[z,\mathtt{j}]$for $z \in IV - \{\mathtt{i}\}$ $invalid[\mathtt{i}] := invalid[\mathtt{j}]$

Table 4: The specialized TVP semantics for calls to `Set` and `Iterator` methods, as well as copy assignments of references to `Set`s and `Iterator`s, illustrate how the instrumentation predicates shown in Table 3 can be updated.

State before the execution of statement 5.			
$invalid[\mathtt{i1}] = 0$	$invalid[\mathtt{i2}] = 0$	$invalid[\mathtt{i3}] = 0$	$sameSet[\mathtt{v},\mathtt{v}] = 1$
$iteratesOver[\mathtt{i1},\mathtt{v}] = 1$	$mutex[\mathtt{i1},\mathtt{i1}] = 0$	$mutex[\mathtt{i1},\mathtt{i2}] = 1$	$mutex[\mathtt{i1},\mathtt{i3}] = 0$
$iteratesOver[\mathtt{i2},\mathtt{v}] = 1$	$mutex[\mathtt{i2},\mathtt{i1}] = 1$	$mutex[\mathtt{i2},\mathtt{i2}] = 0$	$mutex[\mathtt{i2},\mathtt{i3}] = 1$
$iteratesOver[\mathtt{i3},\mathtt{v}] = 1$	$mutex[\mathtt{i3},\mathtt{i1}] = 0$	$mutex[\mathtt{i3},\mathtt{i2}] = 1$	$mutex[\mathtt{i3},\mathtt{i3}] = 0$
State after the execution of statement 5.			
$invalid[\mathtt{i1}] = 0$	$invalid[\mathtt{i2}] = 1$	$invalid[\mathtt{i3}] = 0$	$sameSet[\mathtt{v},\mathtt{v}] = 1$
$iteratesOver[\mathtt{i1},\mathtt{v}] = 1$	$mutex[\mathtt{i1},\mathtt{i1}] = 0$	$mutex[\mathtt{i1},\mathtt{i2}] = 1$	$mutex[\mathtt{i1},\mathtt{i3}] = 0$
$iteratesOver[\mathtt{i2},\mathtt{v}] = 1$	$mutex[\mathtt{i2},\mathtt{i1}] = 1$	$mutex[\mathtt{i2},\mathtt{i2}] = 0$	$mutex[\mathtt{i2},\mathtt{i3}] = 1$
$iteratesOver[\mathtt{i3},\mathtt{v}] = 1$	$mutex[\mathtt{i3},\mathtt{i1}] = 0$	$mutex[\mathtt{i3},\mathtt{i2}] = 1$	$mutex[\mathtt{i3},\mathtt{i3}] = 0$

Figure 7: An illustration of the abstract state maintained by SCMP.

in fact, a member of the finite distributive subset (FDS) framework [56] and the precise solution to the problem can be computed in time $O(EV^2)$ time, where E denotes the number of edges in the control-flow graph of the program in the intraprocedural case (*i.e.*, when the client code consists of a single procedure containing calls to the component methods).

There are some subtle issues in generalizing SCMP to the interprocedural case. Section 6 outlines these issues and shows how SCMP can be solved precisely, context sensitively, in polynomial time in the interprocedural case.

4.3 Extending SCMP to CMP

Note that SCMP focuses on analyzing the heap that is *internal* to `Set` and `Iterator` components, but does not address the question of how to analyze the heap that is *external*, namely the *client*'s heap which may contain objects containing references to collections and iterators. Since we want to formulate an analysis without access to the client code (or a specification of the client code), we must, of necessity, use a generic heap analysis for handling the client's heap. In this section we show how our SCMP algorithm can be extended to handle the client's heap using any one of several different heap analyses. We show how to "lift" instrumentation predicates defined on (statically fixed) variables to the case where such variables reside in the heap (as fields of objects). Though we illustrate this technique for CMP, it is generic and can always be used for such "lifting".

Let *IF* and *VF* denote the set of object fields of type `Iterator` and `Set` respectively. We use the set of instrumentation predicates shown in Table 5. These instrumentation predicates generalize the instrumentation predicates of SCMP (see Table 3) in a very natural way. For instance, recall that SCMP uses a *nullary* predicate *invalid*[`i`] for every iterator variable `i` to track whether the iterator referenced by *variable* `i` is in a valid state or not. Now, we use a *unary* predicate $invalid_h$[`i`](o) for every iterator *field* `i` to track whether the iterator referenced by the `i` field of the object o is in a valid state or not. The other predicates generalized similarly.

The update formulae for the generalized instrumentation predicates can be derived by generalizing the update formulae of Table 4. Table 6 illustrates the generalized update formulae for some of the `Set` and `Iterator` methods. The update formulae for the remaining methods obtained in a very similar fashion.

We refer to the resulting algorithm as HCMP. This is parametric with respect to heap analysis used for the client code, as illustrated by the update formulae in Table 6 which utilize "points-to" information, as represented by predicates of the form *pt*[`x`], for the client's heap. By providing appropriate update formulae for these predicates, ([60] shows how different update formulae can be used to achieve several well known heap analysis techniques) we can generate particular instantiations of HCMP.

Our algorithm is also interprocedural, and uses the techniques outlined in [57] to deal with procedures.

5 Empirical Results

We have prototyped several variants of the HCMP algorithm using the Soot [64] and TVLA [46] frameworks, with the goal of evaluating the feasibility of the algorithm on programs of varying complexity. We wished both to evaluate the precision of the algorithm (in terms of the number of false alarms produced), and to understand the cost/precision tradeoffs arising from various approaches to heap and interprocedual analysis of the client.

The implementation is still at an early stage, and currently does not address multithreading or recursion (since our benchmark programs did not require it). However, both can be handled in TVLA using pre-existing techniques [69, 57].

5.1 Engineering Aspects of the Implementation

Here, we list some of the engineering aspects of our implementation that are material to our results:

In order to reduce the number of potential nodes tracked during analysis, we perform the following preprocessing steps on the program: (i) The `Pack Local Variables` Soot optimization was used to reduce the number of temporary variables; (ii) Soot liveness information is used to assign local reference variables the value `null` when they become dead. The latter optimization is common in partial evaluation and model checking algorithms, e.g., see [6].

We take advantage of Soot's implementation of *Class Hierarchy Analysis* [15] to conservatively construct a method call graph. In addition, Soot treats exceptions by constructing a control flow edge from each statement into all potential corresponding handlers.

We note that the current implementation can lose precision in situations where typing information is lost, (e.g., when collection or iterator-typed objects are cast to `Object`), or when such objects are passed in and out of library methods outside the scope of the analysis.

5.2 Analysis Design Tradeoffs

We experimented with eight variants of HCMP, obtained by considering all possible points in the following design space for client code analysis:

Predicate	Defining TVP formula
$invalid_h[\mathtt{i}](e)$	$\exists o, s, v : rv[\mathtt{i}](e, o) \wedge rv[\mathtt{set}](o, s) \wedge rv[\mathtt{defVer}](o, v) \not\Leftrightarrow rv[\mathtt{ver}](s, v)$
$iteratesOver_h[\mathtt{i}, \mathtt{v}](e_1, e_2)$	$\exists o_1, o_2 : rv[\mathtt{i}](e_1, o_1) \wedge rv[\mathtt{v}](e_2, o_2) \wedge rv[\mathtt{set}](o_1, o_2)$
$mutex_h[\mathtt{i}, \mathtt{j}](e_1, e_2)$	$\exists \quad o_1, o_2, o : rv[\mathtt{i}](e_1, o_1) \wedge rv[\mathtt{j}](e_2, o_2) \wedge$ $o_1 \neq o_2 \wedge rv[\mathtt{set}](o_1, o) \wedge rv[\mathtt{set}](o_2, o)$
$sameSet_h[\mathtt{v}, \mathtt{w}](e_1, e_2)$	$\exists s : rv[\mathtt{v}](e_1, s) \wedge rv[\mathtt{w}](e_2, s)$

Table 5: The modified instrumentation predicates used for HCMP. Note that `i`, `j`, `v`, and `w` now range over *fields* of type `Iterator` or `Set` as appropriate.

Method Call	TVP Semantics
`x.v = new Set()`	$pt[x](e_1) \vee pt[y](e_2) \rhd sameSet_h[\mathtt{v}, \mathtt{v}](e_1, e_2) := (e_1 = e_2)$ $pt[x](e_1) \rhd sameSet_h[\mathtt{v}, z](e_1, e_2) := 0$ for $z \in VF - \{\mathtt{v}\}$ $pt[x](e_2) \rhd sameSet_h[z, \mathtt{v}](e_1, e_2) := 0$ for $z \in VF - \{\mathtt{v}\}$ $pt[x](e_2) \rhd iteratesOver_h[z, \mathtt{v}](e_1, e_2) := 0$ for $z \in IV$
`x.v.add()`	$(\exists e_2 : pt[\mathtt{x}](e_2) \wedge iteratesOver_h[\mathtt{i}, \mathtt{v}](e_1, e_2)) \rhd invalid_h[\mathtt{i}](e_1) := 1$ for $\mathtt{i} \in IF$
`x.i.remove()`	$(\exists e_2 : pt[\mathtt{x}](e_2) \wedge mutex[\mathtt{j}, \mathtt{i}](e_1, e_2)) \rhd invalid_h[\mathtt{j}](e_1) := 1$ for $\mathtt{j} \in IF$

Table 6: Update formulae for the instrumentation predicates of HCMP shown in Table 5.

1. Context sensitive versus context insensitive treatment of client method calls.
2. Using allocation sites versus variables names to distinguish client heap cells. Each node in a TVLA structure represents one or more heap-allocated objects of the client program. The variable names approach [66, 59] merges two heap nodes if the set of variables and fields pointed to them are the same (this abstraction is similar to the one described in Section 3.2). The allocation site approach merges two nodes if they are allocated at the same allocation site in the client program [10, 39].
3. Using a relational versus independent attribute approach for modeling TVLA structures at every program point.

Our empirical observations were as follows:

- The context sensitive algorithms performed better than the corresponding context insensitive ones with respect to *both efficiency and precision.*
- The variables names approach yielded more precise results than the allocation site approach.
- The independent attribute approach was as precise as the relational one and usually faster, which is a result of the precision gained by using instrumentation predicates.

In the interest of space, we give benchmark results only for the context-sensitive, variables-names based, independent attribute variant of the algorithm.

5.3 Benchmarks

Though it is not a highly-tuned implementation, our analysis currently runs quite fast on the benchmark programs. It also produces only 2 false alarms, which we find encouraging, since our analysis is sound.

Table 7 describes the benchmark Java programs; they are publicly available at [67]. The benchmarks use JCF intensively. The *kernel* benchmarks, designed to "stress test" the analysis, include examples illustrating various difficult aspects of CMP as well as typical representative uses of iterators and collections in real applications.

Table 8 displays the results of the experiments. All the experiments were performed on a machine with a 1 Ghz Pentium 3 processor, 1 Gb of memory, running JDK 1.3 Standard Edition on Windows 2000. (Note that the actual memory used by the algorithm ranged from 1 Mb to 71 Mb.)

We found two false alarms in our tests. The first is due to the fact that Java libraries outside the scope of the analysis are not modeled by the analysis. The other false alarm results from not modelling all JCF exceptions. The MapTest benchmark raises the `IllegalStateException` exception when the `remove` method is invoked (although a CME exception would we raised in the absence of the `IllegalStateException` exception). Since our algorithm does not currently model the `IllegalStateException` exception it reports a potential CME which we consider as a false alarm.

Benchmark	**LOC**	**CFG**	**Func**	**Class**	**Description**
Kernel1	128	118	3	1	A set of SCMP examples
Kernel3	192	422	3	3	Collection classes are stored inside heap-allocated lined list cells
Kernel4	166	1103	10	3	Accessing the cells indirectly through auxiliary method calls
Kernel5	55	129	10	2	Figure 1
Kernel6	35	32	3	1	Figure 4
Kernel7	36	36	3	1	Conditional statements manipulating iterators using pointers
MapTest	330	316	7	2	Intensive usage of HashMap and TreeMap implementations
IteratorTest	126	144	10	3	Test program for lots of collections
MapDemo	33	29	3	1	Uses maps for inserts
JFE	2394	2636	35	1	A Java to tvp front end
VCMPKiller	67	155	5	2	A program that stores collections deep in the heap

Table 7: The analyzed programs. Kernel benchmarks and a `VCMPKiller` examples are artificial programs created to illustrate interesting cases of JCC uses. `MapTest` is from [41]. `IteratorTest` and `MapDemo` are examples from [68]. `JFE` is our own Java front used to in our implementation. `LOC` is the number of source lines. `CFG` is the number of control flow nodes in the TVP file (the number of CFG nodes in whole program is actually bigger). `Func` and `Class` are the number of functions and classes, respectively.

Benchmark	**CPU Time (sec)**	**Space (Mb)**	**Struct**	**ERR**	**FA**
Kernel1	0.047	1.8	143	3	0
Kernel3	1.266	5.2	848	3	0
Kernel4	9.625	7.4	2794	3	0
Kernel5	0.125	2.6	187	1	0
Kernel6	0.001	2	29	2	0
Kernel7	0.001	1.9	31	0	0
MapTest	69.17	18.7	4851	0	1
IteratorTest	0.141	3.25	208	0	0
MapDemo	0.015	1.9	26	0	0
JFE	408	71	9890	0	1
VCMPKiller	0.531	4.43	229	0	0

Table 8: The benchmark results of the analyzed programs. **Time** is measured in seconds, **Space** in megabytes. **Struct** is the total number of different structures that arise during the analysis. **ERR** represent the number of real errors found in the benchmarks. Finally, **FA** is the number of false alarms reported by our algorithm.

6 Interprocedural SCMP

Earlier we saw that a precise solution to SCMP can be computed in polynomial time in the intraprocedural case. In this section we show how a precise and context-sensitive solution can be computed in polynomial time for interprocedural SCMP.

Global variables: We observed earlier that our SCMP analysis is an instance of the FDS framework [56]. In the absence of local variables, the techniques presented in [56] can be used in a straighforward fashion to solve interprocedural SCMP.

Local variables: Consider a local variable `v`, of type iterator, of a procedure `P`. Assume that procedure `P` calls, either directly or transitively, a procedure `Q`. The execution of `Q` can not affect the value of local variables, such as `v`, of another procedure. However, the execution of `Q` *can* affect the value of instrumentation predicates, such as *invalid*[`v`], defined on local variables of another procedure. The key issue in interprocedural analysis is to account for such effects.

In the absence of recursion, local variables may be modelled just like global variables, and the IFDS techniques are still applicable. Recursion, however, complicates issues by introducing a potentially unbounded number of variables and a correspondingly potentially unbounded number of instrumentation predicates. The following example illustrates the problem.

```
/*   */ void P (Set S) {
/* 1 */   Iterator i = S.iterator();
/* 2 */   if (...) {
/* 3 */     P(S);
/* 4 */     i.next();  // will throw CME
```

```
/*    */    } else {
/* 5 */       i.next();  // will not throw CME
/* 6 */       i.remove();
/* 7 */       i.next();  // will not throw CME
/*    */    }
/*    */ }
```

Consider the execution path $1, 2, 3, \underline{1}, \underline{2}, \underline{5}, \underline{6}, \underline{7}, 4$, where $\underline{j}$ denotes the execution of line j in a recursive invocation of P. The modification of the collection in line $\underline{6}$ does not make the iterator created in line $\underline{1}$ invalid, but it does make the iterator created in line 1 (*i.e.*, in the earlier invocation of procedure P) invalid. Consequently, the execution of line 7 will *not* throw CME, but when the recursive invocation terminates, the execution of line 4 in the original invocation *will* throw CME.

The example illustrates two points: (a) The analysis has to distinguish between instrumentation predicates corresponding to different recursive instances of the same local variable to avoid imprecision, and (b) The analysis of a procedure can not ignore instrumentation predicates corresponding to local variable instances of procedures up the call chain, even if "hidden" by subsequent recursive invocations, because the values of these instrumentation predicates *can* change during the execution of the procedure (even though the values of the hidden local variables themselves can not be changed).

We now show how local variables can be handled. We assume, for now, that no global variables are used. The only instrumentation predicate whose value can change without any change in the value of any underlying variable is *invalid*[`v`]. In contrast, the value of other instrumentation predicates, such as *mutex*[`u,v`], do not change as long as the value of the underlying variables `u` and `v` do not change. Consider a callsite in a procedure `P` to a procedure `Q`. For any local variable `v` of `P` we need to know how the call to `Q` will affect the value of the instrumentation predicate *invalid*[`v`]. It turns out that we can efficiently compute summary information for procedure `Q` that enables us to answer these questions.

For any procedure `Q`, let Formals(`Q`) denote the set of formal parameters of procedure `Q` of type `Set` or `Iterator`. We compute summary information MayMod(`Q`) for every procedure `Q` that describes the set of collections that may be modified by the execution of `Q`. More precisely, we compute a quantity MayMod(`Q`) that is a subset of *Formals*(`Q`) such that: a collection variable `S` is in MayMod(`Q`) iff the execution of `Q` may modify the collection `S` (either directly by invoking the `Set::add` method on `S` or by creating an iterator on `S` and by invoking the `Iterator::remove` method on that iterator); an iterator variable `I` is in MayMod(`Q`) iff the execution of `Q` may invoke the `Iterator::remove` method on the iterator `I`. Further, the summary computation assumes that there are no aliases between any of the formal parameters. The information MayMod(`Q`) is very similar to the quantity DMOD defined by Banning [2] and can be computed very efficiently [12]. (The one twist is that the problem we are interested in is side-effect analysis when parameters are passed by value but may be one-level pointers, while the problem addressed by Banning is side-effect analysis when parameters are passed by reference and are non-pointers. This is necessary to account for code such as "`D = C; D.add("a");`" where we need to infer that the collection referred to by `C` will be modified by the code fragment.)

The intraprocedural SCMP algorithm can now be extended to analyze a procedure `P`, utilizing only the summary information for any procedure that `P` calls. In particular, assume that MayMod(`Q`) is $\{\mathtt{FC}_1, \cdots, \mathtt{FC}_i, \mathtt{FI}_1, \cdots, \mathtt{FI}_j\}$, where every $\mathtt{FC}_x$ is of type `Set` and every $\mathtt{FI}_x$ is of type `Iterator`. In effect, any call to a procedure `Q` in procedure `P` may be replaced by the following code, where $\mathtt{AC}_x$ and $\mathtt{AI}_y$ denote the actual parameters corresponding to formal parameters $\mathtt{FC}_x$ and $\mathtt{FI}_y$ respectively:

```
if (...)
   AC_1.add("");
else if (...)
   ...
else if (...)
   AC_i.add("");
else if (...)
   AI_1.remove();
else if (...)
   ...
else if (...)
   AI_j.remove();
```

Standard techniques can be used to analyze the whole program using the above method for analyzing a single procedure. For non-recursive programs, the procedures can be processed in top-down order with respect to the call graph. The initial state (or calling context) for any analyzed procedure is obtained by taking the "meet" of the states at all call sites, with appropriate binding of actuals and formals (*e.g.*, the initial value of the instrumentation predicate *invalid*[`f`] is true iff the value of *invalid*[`a`] is true at atleast one of the call sites, where `a` denotes the actual parameter corresponding to the formal parameter `f`). The same procedure works for recursive programs too, except that we need to iterate over loops in the call graph until we reach a fixed point. Alternatively, we can also utilize the techniques of the IFDS framework [56] instead of iterating over the call graph.

Interaction between local and global variables: Interprocedural analysis becomes more complicated in the presence of both local and global variables. Consider a call site. Let `g1` and `g2` denote global variables, and l1 and l2 denote local variables of the calling procedure. The essence of the problem is that the procedure call may modify the value of a global variable, say g1. This can have the effect of changing the value of instrumentation predicates such as *mutex*[g1,l1]. (Note that the value of *mutex*[l1,l2] can not change.) Hence, we need to compute a more general summary information that will let us determine at every call site the changes to the values of the instrumentation predicates such as *sameSet*[g1,l1], *iteratesOver*[g1,l1], and *mutex*[g1,l1].

The summary information we require for a procedure must allow us to answer the following question: let v_in denote any global variable or formal parameter of the procedure; let v_out denote any global variable; is it possible for the output value (*i.e.*, the value at the end of procedure) of v_out to be the same as the input value of v_in? (If v_out is an iterator variable and v_in is a collection variable, we need to be able answer an equivalent question, namely: is is possible for the output value of v_out to be a (newly created) iterator on the input value of v_in?) As before, this summary information is computed assuming that no aliasing exists between any of the formals or globals.

Computing this kind of summary information is at the heart of interprocedural slicing (and the construction of system dependence graphs) [30] and can be computed using the IFDS techniques [56]. Once this summary information is available, we can solve interprocedural HLCMP using the IFDS framework again. (Replacing the use of the IFDS framework by iteration over the call graph without losing precision, requires a more complex "relational" summary that tells us not just what values each global variable can take at the end of a procedure, but tells us for any pair of global iterator variables what pair of values they can take simultaneously).

7 Complexity Results

7.1 Restricted Alias Analysis

This section presents some upper and lower bounds for certain restricted classes of alias analysis problems that are inspired by the SCMP problem. An upper bound for a significantly restricted class of alias analysis problem might not appear very interesting, in general, but we believe these results are relevant and useful in the context of deriving specialized heap analyses for heap manipulted through a restricted interface. We present no proof of these results but the essential idea behind the upper bound results is to show that the "relevant" state can be reduced to a set of nullary instrumentation predicates of certain size, which can be solved using, *e.g.*, using IFDS techniques [56], much as we did in Section 4.1.

We first introduce some terminology. Let TG denote a (closed) set of record types (with simple types being modelled as records with a single field). We say that P is a program over TG if every pointer variable in P is a pointer to an object of some type in TG. We are interested in understanding the complexity of answering alias queries for programs over a given TG.

We define the type graph of TG to be the graph consisting of a node for each type in TG, and an edge $T_1 \rightarrow T_2$ labelled f for every field f in type T_1 of type pointer to T_2. We define $\text{depth}_{TG}(T)$ to be the number of nodes in the longest path to node T. (Thus, a node with no incoming edges is at depth 1.) We define the depth of any alias query $\alpha = \beta$ for program P to be $\text{depth}_{TG}(T)$, where α and β are of type pointers to T.

It is well known that answering a may-alias query of depth 2 is hard [42, 50]. However, it is worth observing that having multiple levels of pointers by itself does not make alias analysis intractable. Destructive heap update is necessary for intractability. We say that a field f of a type T is immutable if the field f of an object of type T is assigned a value only when the object is constructed, and that it is mutable otherwise. We define the *mutation* length of a path in the type graph to be the number of mutable fields in that path. We extend this definition to define the mutation depth of a type and a query much as we did above for depth.

It turns out that any alias query over TG can be answered precisely in polynomial time if its mutation depth is zero. At first blush, this might not sound useful. But immutable fields are common enough in practice for this to be useful. Functional programming techniques, for instance, utilize only immutable fields. Even imperative data structures often use immutable fields: the Iterator::set field is an example of an immutable field.

Let $\text{size}(TG)$ denote the number of different paths in the type graph of TG. Let vars_P denote the number of pointer variables in a program P. Let E_P denote the number of edges in program P.

Theorem 1 *Any alias (may or must) query q for a program P over TG can be precisely answered in time $O(E_P vars_P{}^2 size(TG)^2)$ if the mutation depth of q is zero. Further, any may or must query consisting of a conjunction or disjunction q of k alias queries of mutation depth zero can be precisely answered in time $O(E_P vars_P{}^{2k} size(TG)^2)$.*

We now consider a class of programs where destructive heap update is allowed only in a restricted fashion, inspired by a characteristic of the SCMP problem. Destructive heap update statements are statements of the form:

$$\alpha.f := rhs.$$

A program is said to have the restricted heap update property if the right-hand side of every destructive heap update statement in the program is a newly constructed object (or a "tree" of newly constructed objects, where each object has pointers only to other newly constructed objects).

Theorem 2 *Must-alias queries of mutation depth 1 can be precisely answered in polynomial time for programs with restricted heap update.*

We now present some lower bounds. These are adaptations of well-known intractability results of alias analysis [50] that focus on the restricted classes of alias analysis problems that can be expressed in terms of mutation depth and restricted heap update and can be proved using a simple adaptation of the proofs in [50].

Theorem 3 *May-alias, must-alias, and must-be-null for queries of mutation depth 1 is PSPACE-hard.*

Theorem 4 *May-alias queries of mutation depth 1 is PSPACE-hard even for programs with restricted heap update.*

Theorem 5 *Must-alias queries of mutation depth 2 is PSPACE-hard even for programs with restricted heap update.*

7.2 Control-Flow Effects of Exceptions

In our discussion of SCMP we made the usual assumption that all paths in the program are executable. The concurrent modification exception, however, does affect control flow. In particular, if we consider the sequence of two statements `S1; S2`, execution of S1 will be followed by execution of S2 only if S1 does not throw the concurrent modification exception. If execution of S1 does throw the exception, execution will continue with the innermost matching exception handler that catches the exception or will terminate program execution if no exception handlers catch the exception. It turns out that accounting for the effect of CME on control-flow precisely is hard:

Theorem 6 *Determining if there exists a path P to a program point p such that execution along path P will not throw the concurrent modification exception is P-SPACE hard even for SCMP instances.*

The proof is based on a simple adaptation of the proofs in [50].

8 Related Work and Conclusions

Disjunctive Completion Our systematic derivation of the SCMP algorithm is similar to the process of *disjunctive completion*[14, 23]. In this process, an arbitrary monotonic transfer function on finite sets is converted into a distributive function by lifting to the powerset domain. To avoid the cost of full relational analysis, only domain elements for minimal sets which contribute to the result are introduced. This process can be done effectively for finite domains.

While our specification-driven process for generating instrumentation predicates is similar in many respects to [23], our process is more broadly applicable, since it works over arbitrary first-order formulae and (in the case of HCMP) infinite domains. The iterated weakest precondition process currently must be carried out by hand. In the future, however, we plan to investigate automating our technique by using a decidable logic, e.g., [36, 3, 7] for the weakest precondition computation and appropriate widenings [14] to ensure termination of the iterative process.

Predicate Abstraction The technique of *predicate abstraction* [25] can be used to convert a program over infinite domains to an abstracted program using only booleans. The resulting finite state program can then be more readily analyzed using a variety of techniques. This technique was recently employed by Ball et al. [1] to verify properties of C programs.

Some aspects of predicate abstraction, such as the use of a weakest precondition computation, are similar to our approach to generating instrumentation. However, our work has three critical differences: First, our instrumentation generation phase is applied only to the component specification, not to the client code. This makes it possible to avoid making approximations that are inevitable when the predicate abstraction process is applied to the client. Second, Ball et al. assume that all heap-related aspects of the client have been eliminated by a preprocessing phase, while our client analysis models the heap during analysis by introducing appropriate abstractions. Finally, they assume that the specification contains no heap references, whereas specifying heap properties is central to our approach.

Despite the fact that our specifications are not limited to finite domains, by limiting our instrumentation generation process to the component specification, we can generate sound and accurate analyses of infinitary properties. We do so by using instrumentation to make the precise distinctions necessary to distinguish the component's heap properties of interest, while allowing approximations of the client's heap.

Finite state-space exploration There are a number of projects in which properties of software (usually expressed in some form of specification language) are checked mechanically through exhaustive exploration of a finite (but usually very large) state space. These approaches differ from ours either in requiring that all infinitary aspects of the state space of the code be abstracted *a priori* into finitary structures prior to analysis [13, 26, 1]; by requiring a finite bound on a portion of the infinite concrete state space to be explored [35, 34]; or by considering only temporal (event ordering) properties of the code [20].

In the Bandera project [13], program slicing and various abstraction techniques such as pointer analysis are used to conservatively construct a finite state model from a Java program. The resulting model is then checked against some property expressed in temporal logic using off-the-shelf *model checkers* [11] such as SPIN [28] or SMV [47]. Projects similar to Bandera include Java PathFinder [26], JCAT [16], and Feaver [29].

Jackson and Vaziri [35] also use finite state-space exploration techniques to verify whether Java programs satisfy a specification written in a first-order language called Alloy [33]. Their approach differs from the model-checking approaches in that they are *incomplete*: rather than conservatively extracting a finite-state abstract model of the source program, they explore the *full* concrete state space of the program up to some finite depth. Jackson and Fekete [34] apply this approach to the Concurrent Modification Problem. Since no abstraction is used in this approach, there are no "false positive" error reports; however, there may be valid program states in which the property is not checked. Our approach does not require that a problem manifest itself after a pre-established number of execution steps. While it is commonly believed that most program errors reveal themselves after a few loop iterations or recursive calls, it is not so clear that there is a simple heuristic for suggesting an upper bound on the number of data structures that need to be generated in order for an error to appear.

The FLAVERS system [20] starts with a finite-state specification of a program property of interest, and a *task flow graph* of the program in which only certain *events* of interest are represented. It then uses dataflow analysis to conservatively determine whether the property holds with respect to the task flow graph. FLAVERS focuses on concurrency-related and temporal properties such as races and deadlocks, and appear to have little capability for modeling data structures.

Theorem Proving There are several systems that verify program properties statically using theorem proving techniques. Some notable examples are ESC-Java [44] and PVS [55]. These systems work well provided the user provides appropriate loop invariants, and pre/post-conditions for procedures. In the absence of such notations (e.g., for various kinds of loops in the client code), they often fail to consider many states.

Our approach relies on offline generation of component-specific analyses to avoid the necessity for client annotations. Also, unlike ESC-Java our approach is *sound* (conservative). It is thus guaranteed to catch all errors, and can even prove the absence of errors.

Interprocedural Shape Analysis Noam Rinetzky [58] proposes improving the efficiency of interprocedural shape analysis of programs manipulating abstract data types by utilizing special, hand-crafted, semantics for the abstract data type and investigates this approach for a linked list data structure in the LEDA library.

Other Bug-Finding Tools There are a number of tools that carry out nontrivial static analysis for the purpose of detecting specific classes of programming errors [21, 22, 32, 18, 65, 19, 62]. While our HCMP algorithm is similar to the work above in that it applies static analysis to the detection of a particular programming bug, we believe that the process by which an efficient analysis algorithm is *derived* in a natural way from a simple specification is unusual.

We believe that the notion of deriving efficient program analyzes by a systematic optimization process is promising. Our preliminary benchmark results show that a new analysis algorithm for CMP generated by this technique may scale well enough to be applicable to realistic applications. We are currently pursuing a large number of engineering refinements to our implementation as well as additional benchmark studies, generalizations of the derivation technique to arbitrary specifications, and additional CANVAS infrastructure work.

Acknowledgments

We would like to thank Deepak Goyal for the discussions we had with him on the problem of component-client conformance certification. We would like to thank Sean McDirmid for introducing us to the Concurrent Modification Problem. We would like to thank the SOOT team and Roman Manevich for the assistance they provided with the implementation of the certifier. We would also like to thank Robert O'Callahan, Nurit Dor, Deepak Goyal, Ran Shaham, Carlos Varela, and Reinhard Wilhelm for their helpful feedback on the paper.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of c programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2001.

[2] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *ACM Symp. on Principals of Programming Languages*, pages 29–41, New York, NY, 1979. ACM Press.

[3] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *Proc. European Symposium on Programming, ESOP'99 , LNCS*, 1999. Available at "http://www.math.tau.ac.il/∼ sagiv/esop99.ps".

[4] Bianca. http://bathroom.bianca.com/shack/bathroom/sub/posts/2001_Mar_03/10061/10061.html, 2001.

[5] Blackdown. http://www.blackdown.org/java-linux/java2-status/README-3D121_01, 2001.

[6] M. Bozga, J.-C. Fernandez, and L. Ghirvu. State space reduction based on live variables analysis. In *SAS*, pages 164–178, 1999.

[7] C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. http://www.dcs.qmw.ac.uk/∼ ohearn/.

[8] Canvas project URL. http://www.research.ibm.com/menage/canvas/.

[9] P. Chan, R. Lee, and D. Kramer. *The Java^{TM} Class Libraries, Second Edition, Vol. 1, Supplement for the Java^{TM} 2 Platform Standard Edition, v1.2*, pages 296–325. Addison-Wesley, 1999.

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

[11] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[12] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 57–66, New York, NY, 1988. ACM Press.

[13] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from java source code. In *Proc. 22nd Intl. Conf. on Software Engineering*, June 2000.

[14] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *ACM Symp. on Principals of Programming Languages*, pages 269–282, New York, NY, 1979. ACM Press.

[15] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. Technical Report TR 94-12-01, Washington University, 1994. Also published in ECOOP'95 conference proceedings.

[16] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Software—Practice and Experience*, 29(7):577–603, 1999.

[17] E. Dijkstra. *A Discipline of programing*. Prentice-Hall, 1976.

[18] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Proc. Seventh Intl. Static Analysis Symp.*, pages 115–134, July 2000.

[19] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Proc. 8th Intl. Static Analysis Symp.*, July 2001.

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

[21] D. Evans. Static detection of dynamic memory errors. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 44–53, Philadelphia, May 1996.

[22] C. Flanagan, M. Flatt, M. Krishnamurthi, S. Weirich, and M. Felleisen. Finding bugs in the web of program invariants. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 23–32, Philadelphia, May 1996.

[23] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, Mar. 2000.

[24] J. Gosling, B. Joy, and G. Steele. *The Java^{TM} Language Specification*. Addison-Wesley, 1996.

[25] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *In Proceedings of the 9th Conference on Computer-Aided Verification (CAV'97)*, Haifa, Israel, June 1997.

[26] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Intl. Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[27] L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.

[28] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, 1997.

[29] G. Holzmann and M. Smith. Software model checking: Extracing verification models from source code. In *Proc. FORTE/PSTV*, November 1999.

[30] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, January 1990.

[31] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *ACM Symp. on Principals of Programming Languages*, pages 128–141, London, January 2001.

[32] Intrinsa Corporation. PREfix automated code reviewer. Available at "http://www.intrinsa.com/", 1999.

[33] D. Jackson. Enforcing design constraints with object logic. In *Proc. Static Analysis Symp.* Springer-Verlag, June 2000. LNCS 1824.

[34] D. Jackson and A. Fekete. Lightweight analysis of object interactions. In *Proc. Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, October 2001.

[35] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proc. Intl. Symp. on Software Testing and Analysis*, Portland, OR, August 2000.

[36] J. L. Jensen, M. E. Jorgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pages 226–234, Las Vegas, June 1997.

[37] JLTools. http://cvs.cs.cornell.edu:12000/ commitlogs/jltools/2000-08, 2001.

[38] N. Jones and S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[39] N. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM Symp. on Principals of Programming Languages*, pages 66–74, New York, NY, 1982. ACM Press.

[40] Kaffe. http://rpmfind.net/tools/ Kaffe/messages/1999/1744.html, 2001.

[41] Kaffe. http://rpmfind.net/tools/Kaffe, 2001.

[42] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *ACM Symp. on Principals of Programming Languages*, pages 93–103, New York, NY, 1991. ACM Press.

[43] G. T. Leavans. The java modeling language (JML). http://www.cs.iastate.edu/~ leavens/JML.html.

[44] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000. Available from http://research.compaq.com/SRC/publications.

[45] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA 2000: Proc. of the Int. Symp. on Software Testing and Analysis*, 2000.

[46] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In Springer, editor, *SAS'00, Static Analysis Symposium*, 2000. Available at http://www.math.tau.ac.il/~rumster.

[47] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[48] Microsoft Research. The SLAM project. http://research.microsoft.com/slam/, 2001.

[49] C. Morgan. *Programing from Specifications*. Prentice-Hall, Engelwood N.J, 1990.

[50] R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analyses. In *ACM Symp. on Principals of Programming Languages*, pages 67–80, New York, NY, 2000. ACM Press.

[51] E. Myers. A precise inter-procedural data flow algorithm. In *ACM Symp. on Principals of Programming Languages*, pages 219–230, New York, NY, 1981. ACM Press.

[52] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2001.

[53] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, 1992.

[54] Objectweb. http://www.objectweb.org/messages /EjbContainerGroup/2001/03/msg00028.html, 2001.

[55] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proc. 11th. Intl. Conf. on Automated Deduction*. Springer-Verlag, 1992. LNCS 607.

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

[57] N. Rinetskey and M. Sagiv. Interprocedural shape analysis for recursive programs. In R. Wilhelm, editor, *Proc. of CC 2001*, volume 2027 of *LNCS*, pages 133–149. Springer, 2001.

[58] N. Rinetzky. *Interprocedural Shape Analysis*. PhD thesis, Technion – Israel Institute of Technology, Haifa, Israel, Dec. 2000.

[59] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Prog. Lang. Syst.*, 20(1):1–50, Jan. 1998.

[60] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Symp. on Principals of Programming Languages*, 1999.

[61] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to data flow analysis problems. In P. Deransart and J. Małuszynkski, editors, *LNCS 456, 2nd Workshop on Programming Language Implementation and Logic Programming*. Springer-Verlag, 1990.

[62] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, 2001.

[63] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[64] R. Vallée-Rai, E. Gagnon, L.Hendren, P. Lam, P.Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *9th International Conference on Compiler Construction (CC2000)*, Mar. 2000.

[65] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, 2000.

[66] E. Y.-B. Wang. *Analysis of Recursive Types in an Imperative Language*. PhD thesis, Univ. of Calif., Berkeley, CA, 1994.

[67] A. Warshavsky. http://www.math.tau.ac.il/∼walex, 2001.

[68] M. A. Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, second edition, 2001. http://www.cs.fiu.edu/∼ weiss/dsj2/.

[69] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *ACM Symp. on Principals of Programming Languages*, 2001. http://www.math.tau.ac.il/∼yahave/popl01.ps.