Deriving Specialized Program Analyses for Certifying Component-Client Conformance

G. Ramalingam
IBM T.J. Watson Research Center
rama@watson.ibm.com

Alex Warshavsky
IBM Haifa Research Laboratory and Tel-Aviv University
warshavskyalex@hotmail.com

John Field
IBM T.J. Watson Research Center
jfield@watson.ibm.com

Deepak Goyal
IBM T.J. Watson Research Center
dgoyal@watson.ibm.com

Mooly Sagiv
Tel-Aviv University
sagiv@math.tau.ac.il

ABSTRACT

We are concerned with the problem of statically *certifying* (verifying) whether the client of a software component conforms to the component's constraints for correct usage. We show how conformance certification can be efficiently carried out in a *staged* fashion for certain classes of *first-order safety* (FOS) specifications, which can express relationship requirements among potentially unbounded collections of runtime objects. In the first stage of the certification process, we systematically derive an abstraction that is used to model the component state during analysis of arbitrary clients. In general, the derived abstraction will utilize first-order *predicates*, rather than the propositions often used by model checkers. In the second stage, the generated abstraction is incorporated into a static analysis engine to produce a *certifier*. In the final stage, the resulting certifier is applied to a client to conservatively determine whether the client violates the component's constraints. Unlike verification approaches that analyze a specification and client code together, our technique can take advantage of computationally-intensive symbolic techniques during the abstraction generation phase, without affecting the performance of client analysis. Using as a running example the *Concurrent Modification Problem* (CMP), which arises when certain classes defined by the Java Collections Framework are misused, we describe several different classes of certifiers with varying time/space/precision tradeoffs. Of particular note are precise, polynomial-time, flow- and context-sensitive certifiers for certain classes of FOS specifications and client programs. Finally, we evaluate a prototype implementation of a certifier for CMP on a variety of test programs. The results of the evaluation show that our approach, though conservative, yields very few "false alarms," with acceptable performance.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

General Terms

Verification, Algorithms, Theory, Experimentation, Languages

Keywords

Abstract interpretation, model checking, software components, predicate abstraction, static analysis

1. INTRODUCTION

A fundamental impediment to effective use of software components or libraries is ensuring that client code satisfies the *constraints* that the component imposes as a prerequisite to correct usage. The Canvas[1] project at IBM Research and Tel-Aviv University [4] aims to ease the use of software components 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 paper, we consider a "component" to be any object-oriented software library, and focus on components written in Java.

1.1 The Concurrent Modification Problem

The *Concurrent Modification Problem* (CMP), which arises in the context of the Java Collections Framework (JCF) [5], is a typical conformance constraint problem that we will use as a running example in the sequel. JCF `Iterator`s are used to iterate over the contents of an underlying collection (e.g., a `HashSet`, which implements the `Set` interface). A fundamental constraint on the use of iterators is that once an iterator object o_i is created for 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 restriction 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 throw

[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(); ... }
   void processWorklist() {
     HashSet 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 {
   HashSet s;
   public Worklist() { s = new HashSet(); ...  }
   public void addItem(Object item) { s.add(item); }
   public HashSet unprocessedItems() { return s; }
}
```

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

`ConcurrentModificationException` (or `CME`) when it occurs (note that the name of the exception is misleading, since it often occurs in single-threaded programs). We will use "CMP" to refer to the problem of *statically* determining whether a JCF client may cause `CME` to be thrown.

Consider the Java code fragment in Fig. 1. Here, an iterator is created on a `worklist`, which is implemented using a `HashSet`, a JCF collection class. The iterator is then used to process each item on the worklist in succession. We observe that `CME` can be thrown during item processing, since the nested call to `doSubproblem(...)` causes `worklist.addItem(newitem)` to be called, which will in turn 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.

1.2 First Order Safety Constraints

In Section 2, we will see that CMP is an instance of a class of conformance constraint problems where the component's dynamic state can be characterized (for specification purposes) by a *first-order structure*. A first-order structure consists of a finite collection of *predicates* ranging over a potentially unbounded universe of *individuals*. Using first-order structures to model component state permits, e.g., dynamic memory allocation or dynamic thread creation in the component to be modeled in a natural way [34, 40]. State updates in the component are modelled by updates to predicate values; in addition, the universe of individuals may also evolve—expand and contract—as a consequence of dynamic resource allocation or deletion in the component.

In the sequel, we will develop static certification techniques for certain classes of *first-order safety* (FOS) constraints. In its most general form, an FOS constraint is simply a first-order logical formula φ over the set of predicates characterizing the component state. A client is said to *satisfy* φ if, for any input to the client, every component state encountered during the client's execution satisfies φ. Note that FOS constraints can be trivially used to specify finite state properties of a component, as well as constraints involving dynamically-allocated resources.

In Section 2, we will show how CMP and other FOS constraints can be formally, yet naturally, specified as an abstract program in a language called Easl[2].

1.3 Staged Certification

Our approach to static component certification is based on abstract interpretation [11] of the client program and is conservative, which ensures that every potential violation of the component's conformance constraints is detected. However, a potential drawback of a conservative approach is the possibility of "false alarms": spurious reporting of conformance violations that do not actually occur. For an abstract interpretation to yield acceptable results, it is critical to use a static abstraction of the runtime state that is precise enough to avoid excessive false alarms, without yielding an analysis that is computationally intractable. In this paper, we address this issue by systematically generating component-specific abstractions from a formal specification of a component's conformance constraints in the following *staged* manner:

1. First, the component designer or implementer writes a specification describing both the component's conformance constraints and those aspects of the component's behavior that are pertinent to the component state referred to by the constraints.

2. The component's conformance specification is used to systematically derive a component-specific state abstraction. This abstraction utilizes *instrumentation predicates* defined by first-order logical formulae over the first-order structure underlying the component specification. We refer to such an abstraction as a *first-order predicate abstraction*, since it generalizes the *nullary predicate abstractions* (i.e., proposition) traditionally used in model checking.

3. The generated abstraction is combined with a static analysis engine to yield a *certifier* specific to the component's conformance specification. By choosing between different analysis engines, it is possible to obtain certifiers with various time/space/precision tradeoffs.

4. The certifier is used to analyze the client to conservatively determine whether possible violations of the conformance constraints can occur.

The separation of component-specific abstraction derivation from client analysis is a distinguishing feature of our approach. Since the abstraction derivation process is carried out only at "certifier generation time," resource-intensive symbolic analysis may be used during the abstraction derivation phase without affecting the efficiency of client analysis.

1.4 Overview

In the remainder of the paper, we present the following results:

In Section 2, we show by example how FOS properties such as CMP can be naturally specified as *abstract programs* in Easl.

In Section 3, we show how a simple form of conformance certification can be carried out by applying classical abstract interpretation or dataflow analysis to a composite program formed by inlining the component's behavior specification at every component method call site in the client. For CMP, this certification approach can be viewed as an application of generic heap analysis. However, we show through examples that generic heap analysis (or similar generic program analysis) applied to a composite program tend to be imprecise and/or expensive, since the abstractions used by the

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

generic analysis are oblivious to the details of the constraint specification.

In Section 4, we show how to remedy the imprecision of the generic certification technique using the staged approach outlined in Section 1.3. This process is illustrated for a single CMP client method in which references to collections and iterators are stored only in local or static variables (rather than in object fields). We will refer to this restricted version of CMP as SCMP ("shallow" CMP). We show that this approach yields a simple polynomial-time certifier for SCMP that is *precise*: i.e., the certifier computes the precise meet-over-all-paths solution, and any imprecision in the certifier arises solely from the imprecision in the abstraction used for the *client's* state.

Section 5 shows how to handle *unrestricted* clients of CMP by using a more general version of the abstraction generation process described in Section 4.

Section 6 shows that the staged certification process generates precise certifiers for any member of a class of *mutation-restricted* constraint specifications

In Section 7, we measure the precision and running times of a prototype implementation of our algorithm for the unrestricted CMP problem on a suite of test cases, including both "real-world" programs that use JCF and contrived test cases representing "difficult" instances of CMP. We discuss several configuration options for the core TVLA analysis engine that yield different time/space/precision tradeoffs for CMP analysis. The resulting analysis produces minimal "false alarms" on the test cases, with reasonable speed.

Section 8 shows how the intraprocedural analysis for SCMP presented in Section 4 can be extended to yield a precise, context-sensitive, polynomial-time *inter*procedural analysis for SCMP.

Finally, a detailed comparison to related work is given in Section 9.

2. ABSTRACT PROGRAMS AS SPECIFICATIONS

Easl specifications take the form of *abstract* Java programs (similar facilities are found in JML [24]), which serve both to describe critical aspects of the component's behavior, and to indicate constraints that must be satisfied by any well-behaved client. Easl combines a restricted subset of Java statements (assignments, conditionals, loops, and heap allocation), a restricted set of primitive types (booleans and object references), built-in set and map types, and a *requires* statement (which describes a constraint that must be satisfied at a particular point in the component's execution). These constructs are sufficient to simulate conventional pre- and post-conditions, as well as various forms of finite state specifications. Most importantly, Easl allows a natural expression of component behaviors that determine the relationships among potentially unbounded numbers of component objects.

2.1 Specifying CMP

Fig. 2 contains an Easl specification of CMP. In this specification, every modification of a collection creates a distinct version of that collection, identified by a unique `Version` object. Every iterator records which version of which collection it is associated with. Every use of an iterator is checked for correctness by comparing the version of the iterator with the version of the underlying collection. Note that an update to a collection through an iterator updates both the collection's and the iterator's version, thus ensuring that that particular iterator may continue to be used safely.

```
class Version { /* represents distinct versions of a Set */ }
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;
   Version defVer;
   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 2: An Easl specification of CMP.

The dynamic check for CMP implemented in JCF uses integer-typed versions, rather than heap-allocated version objects, but is otherwise similar to the Easl specification.

We will use the Java code fragment in Fig. 3 as a running example of a CMP client. As explained by the comments in the figure, `CME` may be thrown during the execution of lines `6` or `9`, but not during the execution of line `7`. An analysis that misses the errors in lines `6` or `9` is unsound, while a report of a possible error in line `7` constitutes a false alarm.

2.2 Other FOS Conformance Problems

The following conformance constraint problems are similar to CMP in that they can be naturally modelled by FOS specifications constraining the relationship among instances of dynamically-allocated component objects. All of these problems are members of the class of *mutation-restricted* conformance constraint problems described in Section 6, and admit precise analysis using our staged certification techniques.

Grabbed Resource Problem (GRP): Consider a graph library that provides graph traversal utilities. An implementation of graph traversals where state is stored in vertices (e.g., to record if the vertex has been already visited) does not allow for multiple simultaneous traversals of a graph. Initiating a new traversal of a graph *invalidates* prior traversals of the *same* graph, which may not later be resumed. This is an instance of a general resource-sharing protocol where a resource may be preemptively acquired, which places the constraint that the prior holder may no longer use the resource.

Implementation Mismatch Problem (IMP): Interfaces of modules consisting of multiple interacting types often utilize methods with multiple arguments, with an implicit requirement that the actual arguments all belong to the *same* module implementation. This is the case with the well-known *Factory* design pattern [16].

Alien Object Problem (AOP): A compound object (e.g., a graph) may have a method (e.g., to add an edge) with a restriction that the actual parameters (e.g., of type vertex) "belong" to the graph. Passing a vertex of one graph as an argument to another graph's method could have unintended consequences.

3. GENERIC CERTIFICATION

Since the Easl specification of CMP in Fig. 2 models the component behavior via manipulation of an "abstract heap," it is natural to consider whether certification of this or similar problems could be carried out using generic heap analysis. Indeed, this is the case. In general, one approach to carrying out certification would be as follows: (a) Create a composite program by combining the client code

```
/* 0 */  Set v = new Set();
/* 1 */  Iterator i1 = v.iterator();
/* 2 */  Iterator i2 = v.iterator();
/* 3 */  Iterator i3 = i1;
/* 4 */  i1.next();
// The following update via i1 invalidates the
// iterator referred to by i2.
/* 5 */  i1.remove();
/* 6 */  if (...) { i2.next(); /* CME thrown */ }
// i3 refers to the same, valid, iterator as i1
/* 7 */  if (...) { i3.next(); /* CME not thrown */ }
// The following invalidates all iterators over v
/* 8 */  v.add("...");
/* 9 */  if (...) { i1.next(); /* CME thrown */ }
```

Figure 3: A Java program fragment illustrating CMP.

and the component specification, treating the specification as the component *implementation*. This is particularly simple to do with Easl specifications, since they take the form of abstract programs. (b) Apply a suitable analysis algorithm to the resulting composite program and verify that whenever any `requires` clause in the specification is executed, the expression in the clause will evaluate to true. In the case of CMP, we could carry out Step (b) using any existing algorithm for *must-alias* analysis, since the `requires` clauses of the CMP specification all entail equality comparisons of two pointer-valued expressions. (This approach can be generalized to *arbitrary* Easl specifications using a generic analysis engine. See [29] for details.)

Let us now consider how well certification based on generic alias analysis works for CMP. Consider, for example, an *allocation-site* based analysis [6] which does not distinguish between different objects allocated at the same program point.

```
Set s = new HashSet();
while (...) {
  s.add(...);
  for (Iterator i = s.iterator(); i.hasNext(); ) {
    Object o = i.next();
  }
}
```

An allocation-site based alias analysis will be unable to certify that this fragment is free of CMP errors, because the analysis will be unable to distinguish between the different *versions* of set `s` inside the loop. Similar problems occur when other generic heap analyses are applied to CMP; e.g., see Section 4.4.

We note that SCMP (CMP restricted to client programs in which references to collections and iterators are stored only in static or local variables) can be seen as a must-alias problem with 3-level pointers—a problem for which precise analysis is in general *PSPACE*-hard in the intraprocedural and *EXPTIME*-hard in the interprocedural case [27]. This is one reason to suspect that even SCMP is likely to be resistant to most generic heap analysis algorithms, and that any generic heap analysis algorithm of sufficient accuracy to be useful for SCMP would likely be quite inefficient.

4. STAGED CERTIFICATION

The problem with using any generic analysis engine for certification is that such an engine must use abstractions based on program properties unrelated to the component conformance constraints of interest. In this section, we show how to remedy this problem by using a component's conformance constraint specification to derive a *specialized* abstraction of the components' state. This abstraction technique will yield more precise and efficient certifiers than the generic abstractions used in Section 3.

We illustrate the staged abstraction process using CMP. We will restrict our attention in this section to CMP clients in which references to collections and iterators are stored only in local or static variables (rather than in object fields). We will refer to this special case of CMP as SCMP. We will also assume in this section that the client contains no calls to other client methods (i.e., that the client analysis is intraprocedural). We will show that the generated abstraction yields a polynomial time certifier that computes the precise meet-over-all-paths solution for (intraprocedural) SCMP.

This section is intended to convey the essential ideas of our approach without excessive formalism. Section 5 fills in most of the formal details, and extends the results to arbitrary client programs by describing how component references in the client heap are handled, and how constructs such as client method calls are treated. In addition, Section 8 describes a precise, polynomial-time *interprocedural* certifier for the special case of SCMP.

Our component abstraction process will consist of: (a) *component state abstraction*, which characterizes those aspects of the component state (more precisely, the state of all component object instances) that are relevant to the certification process, and (b) *component method abstraction*, which identifies how the component state abstraction is updated as a result of a component method call.

4.1 Deriving Component State Abstraction

We will represent the relevant state of a component using *instrumentation predicates* [34], which can be viewed as refining the component's state with derived information specific to the certification problem. Given the SCMP restriction, the abstraction derivation process yields a *nullary predicate abstraction*, i.e., a collection of nullary predicates (or propositions, or boolean variables). In Section 5, we will describe a more general abstraction derivation process that can yield predicates of arbitrary arity (a *first-order predicate abstraction*). For the sake of simplicity, we will assume that the component specification contains `requires` clauses only at method entry. Component specifications that contain `requires` clauses at points other than at method entry may be handled by suitably generalizing the approach below.

We identify instrumentation predicates by iteratively performing a symbolic, backward weakest-precondition [14] computation over every possible sequence of component method calls, using the following rules:

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 (where none of the φ_i is a disjunct of the form $\alpha \vee \beta$), 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`, $WP(\mathsf{S}, \varphi)$, is a *candidate instrumentation formula*. ($WP(\mathsf{S}, \varphi)$ is a formula that holds before the execution of `S` iff φ holds after the execution of `S`.)

The motivation for rules 1 and 3 should be clear. The motivation for rule 2 is slightly more complex. The specific form of rule 2 is intended to enable the use of an efficient *independent attribute* [28, p. 247] analysis without losing the precision of *relational* analysis [28, p. 248]. If $\varphi_1 \vee \varphi_2$ is a candidate instrumentation formula, tracking the values of φ_1 and φ_2 separately does not lead to a loss of precision.

Predicate	Meaning (in Easl)
$stale_{\texttt{i}}$	`i.defVer != i.set.ver`
$iterof_{\texttt{i,v}}$	`i.set == v`
$mutx_{\texttt{i,j}}$	`(i.set == j.set) && (i!= j)`
$same_{\texttt{v,w}}$	`v == w`

Figure 4: The instrumentation predicates used for component state abstraction in CMP.

In contrast, if $\varphi_1 \wedge \varphi_2$ is a candidate instrumentation formula, then we create a *single* instrumentation predicate $\varphi_1 \wedge \varphi_2$ since tracking the values of φ_1 and φ_2 separately may lead to imprecision when used with an independent attribute analysis. This is related to the notion of disjunctive completion [10, 17] and distributivity in static analysis.

Component State Abstraction for CMP

We now illustrate the process above by applying it to the specification of CMP.

Step 1: 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.ver` may be true. We therefore introduce a new predicate $stale_{\texttt{i}}$ to represent the formula `i.defVer != i.set.ver`.

Step 2: Next, we consider how the execution of different `Set` and `Iterator` methods affect the value of predicate $stale_{\texttt{i}}$. Consider the execution of a method call `v.add()`, where `v` is of type `Set`. $stale_i$ is true after the execution of `v.add()` iff the condition ($stale_{\texttt{i}}$ || (`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 of $stale_{\texttt{i}}$. Hence, we introduce a second instrumentation predicate $iterof_{\texttt{i,v}}$, representing the condition `i.set == v`.

Step 3: Consider the effect of executing `j.remove()` (where `j` is an iterator variable) on predicate $stale_{\texttt{i}}$. It can be verified that $stale_{\texttt{i}}$ is true after execution of `j.remove()` iff the condition ($stale_{\texttt{i}}$ || (`(i.set == j.set) && (i != j)`)) is true before the execution of the statement. We introduce the instrumentation predicate $mutx_{\texttt{i,j}}$, representing the condition `(i.set == j.set) && (i != j)`.

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

We have now reached a fixed point: applying the rules for identifying instrumentation predicates will not produce any more predicates. Fig. 4 presents the definitions of the instrumentation predicates identified.

Predicate Families

Our earlier description of the rules for identifying instrumentation predicates omitted certain details for the sake of clarity. An expression identified as a candidate instrumentation predicate by these rules will, in general, contain free variables. As an example, the expression `i.defVer != i.set.ver` identified as being a candidate instrumentation predicate contains a free variable `i`. Such an expression really identifies a *family* of instrumentation predicates for a given client program which contains one predicate for every variable in the client program whose type is the same as that of the free variable (here, `Iterator`). Specifically, let *I* and *V* denote respectively the set of `Iterator` variables and the set of `Set` variables in an SCMP client. The set of predicates we use for analysis of the client is

$$\{\ stale_{\texttt{i}} \mid \texttt{i} \in I\ \} \cup \{\ iterof_{\texttt{i,v}} \mid \texttt{i} \in I, \texttt{v} \in V\ \} \cup \\ \{\ mutx_{\texttt{i,j}} \mid \texttt{i},\texttt{j} \in I\ \} \cup \{\ same_{\texttt{v,w}} \mid \texttt{v},\texttt{w} \in V\ \}.$$

4.2 Deriving Component Method Abstraction

Having described how to derive a component state abstraction, we must now identify the corresponding abstraction of the component's collection of methods. Each abstracted method will define how a call to that method affects the values of instrumentation predicates comprising the state abstraction.

The iterative rules of Section 4.1 for deriving a component's state abstraction can also be seen to generate the method abstractions, as follows: Let φ_0 be a candidate instrumentation predicate, and let the weakest precondition of φ with respect to component method `M` be $\varphi_1 \vee \ldots \vee \varphi_k$. Each candidate φ_i is represented in the abstraction by a corresponding (boolean-valued) Easl variable p_i. We define the *update formula* for p_0 for method `M` to be "$p_0 := p_1 \vee \ldots \vee p_k$". The abstraction of method `M` consists of an update formula for every instrumentation predicate, as well as the precondition of the method (expressed as a `requires` clause).

The method abstractions obtained from the CMP specification are shown in Fig. 5. (These abstractions have been optimized by eliminating update formulae of the form $p_0 := p_0$, which correspond to instrumentation predicates whose values are not affected by the method call.) The method abstractions have been presented in a form that reflects the intended use of these abstractions during certification of a given client. In particular, during certification, calls to component methods in the client code (the left column of the table) will be replaced by the corresponding method abstraction (the right column of the table). The same table also presents abstractions of copy assignments of component references, which serve the same purpose as the method abstractions.

The table in Fig. 5 is parametrized by *I* and *V*, which denote respectively the set of `Iterator` variables and the set of `Set` variables in a client program. The method abstractions utilize a macro mechanism, indicated by the $\forall$ quantifier. In particular, a macro-based update of the form "`lhs(z) := rhs(z)` $\forall$ `z` $\in$ `S`" represents a whole set of updates, one for each `z` $\in$ `S`.

4.3 Specialized Certification

Let us now see how the component and method abstractions are used in certification of a client program. The first step in the certification process is to *transform* the client program by (a) replacing variables in the program that are references to the component by boolean variables corresponding to the identified instrumentation predicates, and (b) replacing calls to component methods by a corresponding instantiation of the identified method abstraction. (The part of the transformed client program that relates to the component resembles a *boolean program* [2].)

To determine whether the inlined component method preconditions will always be satisfied, we use standard program analysis techniques. We note that the transformed program has a very special form: each assignment statement is of the form $p_0 := p_1 \vee \ldots \vee p_k$ or $p := 0$ or $p := 1$. As a result, any fixed point computation for a set of distributive equations over values in the set of subsets of $\{0, 1\}$ (e.g., finite distributive subset (FDS) analysis [30]) can be used to compute the possible values of each variable at every program point in the transformed program.

Statement	Specialized Abstraction
`v = new Set()`	$same_{\mathtt{v},\mathtt{v}} := 1$ $same_{\mathtt{v},\mathtt{z}} := 0 \quad \forall \mathtt{z} \in V - \{\mathtt{v}\}$ $same_{\mathtt{z},\mathtt{v}} := 0 \quad \forall \mathtt{z} \in V - \{\mathtt{v}\}$ $iterof_{\mathtt{k},\mathtt{v}} := 0 \quad \forall \mathtt{k} \in I$
`v.add()`	$stale_{\mathtt{k}} := stale_{\mathtt{k}} \vee iterof_{\mathtt{k},\mathtt{v}} \quad \forall \mathtt{k} \in I$
`i = v.iterator()`	$iterof_{\mathtt{i},\mathtt{z}} := same_{\mathtt{v},\mathtt{z}} \quad \forall \mathtt{z} \in V$ $mutx_{\mathtt{i},\mathtt{i}} := 0$ $mutx_{\mathtt{i},\mathtt{k}} := iterof_{\mathtt{k},\mathtt{v}} \quad \forall \mathtt{k} \in I - \{\mathtt{i}\}$ $mutx_{\mathtt{k},\mathtt{i}} := iterof_{\mathtt{k},\mathtt{v}} \quad \forall \mathtt{k} \in I - \{\mathtt{i}\}$ $stale_{\mathtt{i}} := 0$
`i.remove()`	requires $\neg\, stale_{\mathtt{i}}$ $stale_{\mathtt{j}} := stale_{\mathtt{j}} \vee mutx_{\mathtt{j},\mathtt{i}} \quad \forall \mathtt{j} \in I$
`i.next()`	requires $\neg\, stale_{\mathtt{i}}$
`v = w`	$same_{\mathtt{v},\mathtt{z}} := same_{\mathtt{w},\mathtt{z}} \quad \forall \mathtt{z} \in V - \{\mathtt{v}\}$ $same_{\mathtt{z},\mathtt{v}} := same_{\mathtt{z},\mathtt{w}} \quad \forall \mathtt{z} \in V - \{\mathtt{v}\}$ $iterof_{\mathtt{k},\mathtt{v}} := iterof_{\mathtt{k},\mathtt{w}} \quad \forall \mathtt{k} \in I$
`i = j`	$iterof_{\mathtt{i},\mathtt{z}} := iterof_{\mathtt{j},\mathtt{z}} \quad \forall \mathtt{z} \in V$ $mutx_{\mathtt{i},\mathtt{k}} := mutx_{\mathtt{j},\mathtt{k}} \quad \forall \mathtt{k} \in I - \{\mathtt{i}\}$ $mutx_{\mathtt{k},\mathtt{i}} := mutx_{\mathtt{k},\mathtt{j}} \quad \forall \mathtt{k} \in I - \{\mathtt{i}\}$ $stale_{\mathtt{i}} := stale_{\mathtt{j}}$

Figure 5: The component method abstraction for CMP. Here, `i`, `j`, and `k` denote variables of type `Iterator`, while `v`, `w`, and `z` denote variables of type `Set`.

Since every `requires` clause in the transformed program takes the special form "`requires` $\neg v$", this information is sufficient to perform the certification.

We now illustrate this process for SCMP. The code fragment in Fig. 6 illustrates how selected statements of the CMP example of Fig. 3 are transformed. Note that the declarations of the `Iterator` and `Set` variables are replaced by the declaration of corresponding sets of boolean variables representing the nullary predicate abstraction derived in Section 4.1. The transformation of Statement 5 of the example is particularly notable. Here, we see that the condition in the specification's `requires` clause, which checks the validity of iterator `i1`, is satisfied. However, as a result of executing statement, the value of variable $stale_{i2}$ becomes 1, which will prevent the `requires` clause corresponding to the use of `i2` in Statement 6 (translation omitted) from being satisfied.

The next step is to analyze this transformed program to determine the possible values of the boolean variables mentioned in the `requires` clauses in the transformed program. In the intraprocedural case, the precise (meet-over-all-paths) solution to this problem can be computed in time $O(EB^2)$ time using FDS analysis [30], where B denotes the number of iterator and collection variables in the original program and E denotes the number of edges in the control-flow graph of the program. We address the interprocedural version of the problem in Section 8 and show how the meet-over-all-*valid*-paths solution can also be computed in polynomial time.

4.4 Specialized vs. Generic Abstraction

Fig. 7(a) and Fig. 7(b) depict the *concrete* state of the program of Fig. 3 before and after execution of Statement 5. Fig. 8 depicts the *abstract* states computed by the staged certifier that correspond to those concrete states.

It is interesting to compare the abstract state used by our specialized certifer with that computed by a sophisticated heap analysis based on *storage shape graphs* [37, 33]. These analyses merge

```
// variables representing values of nullary predicate abstraction
// used for certification
boolean $stale_{i1}, stale_{i2}, stale_{i3}$;
boolean $iterof_{i1,v}, iterof_{i2,v}, iterof_{i3,v}$;
boolean $mutx_{i1,i1}, mutx_{i1,i2}, mutx_{i1,i3}, mutx_{i2,i1}, mutx_{i2,i2}$;
boolean $mutx_{i2,i3}, mutx_{i3,i1}, mutx_{i3,i2}, mutx_{i3,i3}$;
boolean $same_{v,v}$;
...
// 0:  Set v = new Set();
  $same_{v,v} := 1$;
  $iterof_{i1,v} := 0$;
  $iterof_{i2,v} := 0$;
  $iterof_{i3,v} := 0$;
// 1:  Iterator i1 = v.iterator();
  $iterof_{i1,v} := same_{v,v}$;
  $mutx_{i1,i1} := 0$;
  $mutx_{i1,i3} := iterof_{i3,v}$;  $mutx_{i3,i1} := iterof_{i3,v}$;
  $mutx_{i1,i2} := iterof_{i2,v}$;  $mutx_{i2,i1} := iterof_{i2,v}$;
  $stale_{i1} := 0$;
// 2:  Iterator i2 = v.iterator();
  $iterof_{i2,v} := same_{v,v}$;
  $mutx_{i2,i2} := 0$;
  $mutx_{i2,i1} := iterof_{i1,v}$;  $mutx_{i1,i2} := iterof_{i1,v}$;
  $mutx_{i2,i3} := iterof_{i3,v}$;  $mutx_{i3,i2} := iterof_{i3,v}$;
  $stale_{i2} := 0$;
...
// 5:  i1.remove();
  requires $\neg stale_{i1}$;   // requires statement is satisfied
  $stale_{i1} := stale_{i1} \vee mutx_{i1,i1}$;
  $stale_{i2} := stale_{i2} \vee mutx_{i2,i1}$;   // $stale_{i2}$ becomes 1
  $stale_{i3} := stale_{i3} \vee mutx_{i3,i1}$;
...
```

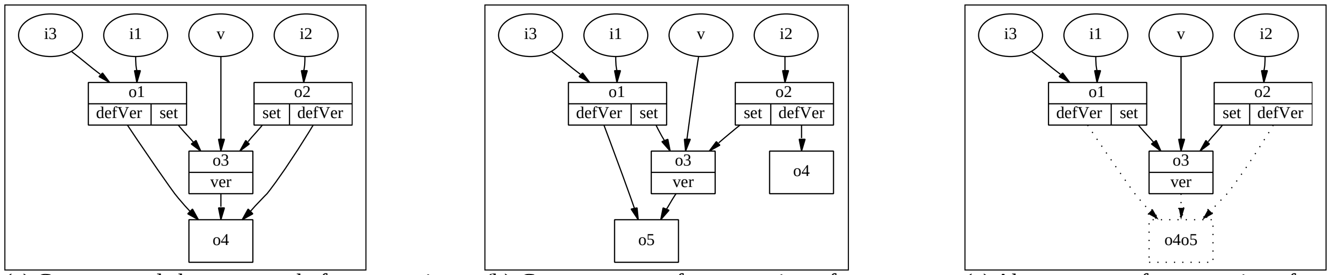
Figure 6: A fragment of the transformed client program of Fig. 3

nodes in a storage shape graph together if and only if they are pointed to by the same set of variables. Let us see what happens at Statement 5 of Fig. 3 when we apply the analysis based on shape graphs. The two concrete nodes for the two version objects $o4$ and $o5$ are merged together because there are no pointer variables that point to either object. 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*. Hence, the analysis will produce a false alarm at statement 7.

Note that the state representation in Fig. 8 is much more compact, yet more precise, than the state representation in Fig. 7(c). In particular, it enables the staged certifier to determine that iterator `i3` is valid after statement 5 and avoid producing a false alarm at statement 7.

4.5 The Derivation Process: Details

We have seen that the derivation procedure for identifying instrumentation predicates converges quickly for CMP. In Section 6, we show that the process converges with a finite set of instrumentation predicates for a class of Easl specifications called *mutation-restricted* specifications. The corresponding abstraction produced for the components is a *finite but precise* abstraction: i.e., any imprecision in a certifier that uses the derived component abstraction is solely due to the imprecision in the abstraction used for the *client's* state. In the general case, however, there is no guarantee that the derivation procedure will terminate. In other words, for some examples, there may be no finite bound on the number of instrumentation predicates generated by this procedure. However, in the general case, heuristics may be used to stop the generation of new instrumentation predicates at some point during the derivation process. This will, in turn, require introducing approximate but conservative method abstractions (since some of precise update formulae required for a method abstraction may not be expressible in terms



(a) Concrete and abstract state before execution of statement 5.

(b) Concrete state after execution of statement 5.

(c) Abstract state after execution of statement 5.

Figure 7: (a) and (b) are storage shape graphs representing the concrete state before and after execution of line 5. (c) is a storage shape graph that abstracts the concrete state depicted by (b). A solid edge denotes a "must" point-to relation and a dotted edge denotes a "may" point-to relation.

of the generated instrumentation predicates). Identifying suitable heuristics to ensure termination of the derivation procedure is beyond the scope of this paper.

The abstraction derivation process entails checking whether a newly generated instrumentation predicate is equivalent to any of the instrumentation predicates already generated. While simple conservative equality checks (such as converting the predicates to a normal form, e.g. disjunctive normal form, and comparing them for syntactic equality) are sufficient to ensure termination of the derivation procedure in examples such as CMP, more powerful decision procedures can be used to make the derivation procedure more effective and to reduce the number of generated instrumentation predicates. While the use of alternative decision procedures for equality has an impact on the set of generated instrumentation predicates, it does not affect the correctness of the approach.

State before the execution of statement 5.		
$stale_{\mathtt{i1}} = 0$	$stale_{\mathtt{i2}} = 0$	$stale_{\mathtt{i3}} = 0$
$iterof_{\mathtt{i1,v}} = 1$	$iterof_{\mathtt{i2,v}} = 1$	$iterof_{\mathtt{i3,v}} = 1$
$mutx_{\mathtt{i1,i2}} = 1$	$mutx_{\mathtt{i1,i3}} = 0$	$mutx_{\mathtt{i2,i3}} = 1$

State after the execution of statement 5.		
$stale_{\mathtt{i1}} = 0$	$stale_{\mathtt{i2}} = 1$	$stale_{\mathtt{i3}} = 0$
$iterof_{\mathtt{i1,v}} = 1$	$iterof_{\mathtt{i2,v}} = 1$	$iterof_{\mathtt{i3,v}} = 1$
$mutx_{\mathtt{i1,i2}} = 1$	$mutx_{\mathtt{i1,i3}} = 0$	$mutx_{\mathtt{i2,i3}} = 1$

Figure 8: An illustration of the abstract state maintained by our specialized certifier for SCMP. Note that the state representation has been simplified using the facts that for all x,y $same_{\mathtt{x,x}} = 1$, $mutx_{\mathtt{x,x}} = 0$, and $mutx_{\mathtt{x,y}} = mutx_{\mathtt{y,x}}$.

4.6 Relation to Predicate Abstraction

The approach described in this section is closely related to the notion of predicate abstraction [18]. The term "predicate abstraction" has been used in the literature to denote both the approximation of a program's state by a collection of boolean variables (nullary predicates), as well as the approximation of a program's statements by statements that refer only to these boolean variables. Recent work on *counterexample-driven abstraction refinement* focuses on improving predicate abstraction based verification by "discovering" predicates that are relevant to the particular verification problem instance [1, 35, 7]. While our goals are similar, our approach is orthogonal to previous approaches. Our predicate discovery algorithm is applied only to a component/library specification, to determine a suitable abstraction of the component. This allows us to *stage* the whole process, as the component specification can be analyzed before any client program is available, and the results (re)used for checking any possible client program. This can serve as the basis for a more *modular and scalable* predicate discovery based verification, and avoid or reduce the need for expensive symbolic analysis during analysis of the client program. In contrast, predicate discovery in prior work is done lazily, during the analysis of a complete program/system, over a single execution path. This, however, has the advantage that it can produce better abstractions. The two approaches are complementary and can be employed together to improve efficiency, while retaining flexibility.

A second difference between our approach and prior work based on predicate abstraction is in how the abstracted program is subsequently analyzed. While prior work relies on model checking techniques whose complexity is exponential (they correspond to relational analysis), our approach is not restricted to using an exponential relational analysis. In fact, our predicate discovery is intended to enable the use of more efficient independent attribute analysis without losing the precision of relational analysis. This is what enables our technique to produce a polynomial time SCMP certifier that computes the precise meet-over-all-paths solution.

A third difference between our approach and prior work is the subject of the next section.

5. FIRST ORDER PREDICATE ABSTRACTION

In Section 4, we saw how a component specification can be used to derive a specialized abstraction which can in turn serve as the basis for more precise certification. For SCMP, the relevant component state was described by a *nullary predicate abstraction*. In the general case, we will utilize *first-order* predicate abstraction, where both nullary and non-nullary predicates may be used to describe the relevant program state. The component abstraction process for SCMP was described in terms of operations on Java-style statements and predicates. However, to describe the first-order abstraction derivation process for arbitrary client programs, we will require a more expressive intermediate language. We will use the TVP [26] intermediate language for this purpose.

In the remainder of this section, we give an overview of the TVP language, show how client programs may be translated into TVP, and generalize the component abstraction process of Section 4 to clients where component references may be stored in the heap. As before, we describe the abstraction process by example using CMP. Finally, we present a brief overview of the TVLA system [26], a configurable abstract interpreter for TVP. We use TVLA as a certification engine for TVP programs formed by combining the TVP translation of a client program with the first-order abstractions generated for the component.

5.1 An Overview of TVP

TVP is an imperative programming language built around an expression sub-language based on first-order logic. The only type of variables allowed in TVP are predicate variables. Program states are represented in TVP by *2-valued logical structures*. A 2-valued structure over a set of predicate variables P is a pair $\langle U, \iota \rangle$, where U is a set referred to as the universe, and ι is a function that maps predicate variables to their values: for every predicate $p \in P$ of arity k, $\iota(p) \colon U^k \to \{0, 1\}$.

A TVP program is a control-flow graph, each edge annotated with an *action*, consisting of:

(a) An optional *precondition* (a first-order logic formula), which is used to model conditional branches.

(b) Zero or more *allocation* bindings of the form "`let id = new() in`", which adds a new element to the universe. The newly-added element may be referred to as `id` in the following assignment; 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.

(c) Zero or more predicate updates of the form $p(v_1, \ldots, v_m) := \varphi(v_1, \ldots, v_m, n_1, \ldots, n_k)$, where each n_i is a variable bound by an allocation binding, which assigns a new value to predicate variable p. We use the notation $guard \rhd lhs := rhs$ as shorthand for "$lhs := (guard \wedge rhs) \vee (\neg guard \wedge lhs)$".

5.2 Standard Translation of Client Programs

We now present a standard way of modelling the state of an arbitrary Java program using a 2-valued structure:

- Every heap-allocated object is modelled by an element of the universe.
- Every (static) 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.
- Every field `fld` of a reference type is represented by a binary predicate $rv_{\texttt{fld}}$; 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.
- TVP provides no explicit support for procedures. Procedures are modelled [31] by explicitly modelling a stack of activation records. In particular, an activation record is modelled by an element of the universe, and local variables of procedures are modelled just like fields: the value of $rv_{\texttt{lv}}(o_1, o_2)$ is true iff the variable `lv` of activation record o_1 refers to object o_2.

Fig. 9 illustrates how typical Java heap manipulation statements will be translated into TVP actions under the above model. (Note that the translation for `x = new C()` models only allocation; initialization of the object by the constructor is modelled separately.) Generic certification (see Section 3) is done by translating both the client program and the component specification into a composite TVP program in this fashion, and by applying abstract interpretation to this composite program. (We will soon explain the specific abstract interpretations we use for TVP.) Code manipulating primitive (non-reference) types is abstracted away (i.e., is not modelled) by the translation.

The above approach, effectively, analyzes the composite program using the heap analysis algorithm described in [37, 33]. Simple variations on the scheme described above for translating Java programs into TVP can be used to utilize several other well known heap analysis techniques for certification (see ([34]).

Java Statement	TVP Translation
`x = new C()`	let n = new() in $pt_{\texttt{x}}(o) := (o = n)$
`x = y`	$pt_{\texttt{x}}(o) := pt_{\texttt{y}}(o)$
`x = y.fld`	$pt_{\texttt{x}}(o) := \exists o_1 : pt_{\texttt{y}}(o_1) \wedge rv_{\texttt{fld}}(o_1, o)$
`x.fld = y`	$pt_{\texttt{x}}(o_1) \rhd rv_{\texttt{fld}}(o_1, o_2) := pt_{\texttt{y}}(o_2)$

Figure 9: Translating the pointer manipulation statements of Java into TVP.

5.3 Specialized Component Abstraction

As explained in Section 4, our approach is to derive a specialized abstraction from the component specification, which serves as the basis for more precise certification. This specialized abstraction takes the form of a set of instrumentation predicates. An instrumentation predicate [34] is defined by a formulae over the predicates of the standard abstraction.

The derivation procedure described in Section 4 produces an abstraction consisting of a set of instrumentation predicate *families*, parametrized over references to the relevant components. (E.g., *stale* is a predicate family parametrized over `Iterator` references.) The actual set of instrumentation predicates used for certification of a given client is obtained by instantiating these families for the given client. We saw in Section 4 how this instantiation is done for a simple case, namely client programs where component references are stored only in static variables (rather than in object fields). We now show how the abstraction is instantiated and used in the general case.

Consider an instrumentation predicate $\phi(x_1, \cdots, x_k)$ with free variables x_1 through x_k of type C, derived from a specification for component C. Let `CF` denote the set of object fields of type C in a given client program. The family of ϕ predicates used for analysis of this client consists of a k-ary predicate $\phi_{f_1, \cdots, f_k}$ for every k-tuple $(f_1, \cdots, f_k)$ in $\texttt{CF}^k$. The value of $\phi_{f_1, \cdots, f_k}(o_1, \cdots, o_k)$ is true iff the k component instances pointed to by field f_1 of object o_1, through field f_k of object o_k, satisfy the property ϕ. (This scheme can also be used to deal with local variables of type C by treating a local variable to be a field of an "activation record object".)

As an example, consider the *stale* family of predicates used for CMP, which is defined by the formula `i.defVer != i.set.ver` with one free variable `i`. For certification of a CMP client, we will use a *unary* predicate $stale_{\texttt{f}}(o)$ for every iterator *field* `f` in the client program to track whether the iterator referenced by the `f` field of the object o is in an invalid state. (In contrast, the SCMP analysis uses a *nullary* predicate $stale_{\texttt{x}}$ for every iterator variable `x` to track whether the iterator referenced by *variable* `x` is invalid.) Fig. 10 illustrates the set of instrumentation predicates used for CMP certification, as well as the definition of these predicates in TVP. (The defining TVP formulae can be obtained from the corresponding `Easl` formulae presented in Fig. 4 using the standard translation from Java to TVP described earlier in this section.)

Next, we consider component method abstraction. Each method abstraction takes the form of a collection of TVP actions. Fig. 11 presents the method abstractions for some of the `Set` and `Iterator` methods for CMP. These are obtained by generalizing the simpler method abstractions presented in Fig. 5 for SCMP. Abstractions for the remaining methods can be obtained similarly.

5.4 Specialized Translation of Client Programs

The specialized component abstraction is used in the translation of a client program into TVP. In particular, component method calls in the client program are translated using the corresponding method

Predicate	Defining TVP formula
$stale_{\mathtt{i}}(e)$	$\exists o, s, v : rv_{\mathtt{i}}(e, o) \wedge rv_{\mathtt{set}}(o, s) \wedge rv_{\mathtt{defVer}}(o, v) \not\Leftrightarrow rv_{\mathtt{ver}}(s, v)$
$iterof_{\mathtt{i},\mathtt{v}}(e_1, e_2)$	$\exists o_1, o_2 : rv_{\mathtt{i}}(e_1, o_1) \wedge rv_{\mathtt{v}}(e_2, o_2) \wedge rv_{\mathtt{set}}(o_1, o_2)$
$mutx_{\mathtt{i},\mathtt{j}}(e_1, e_2)$	$\exists o_1, o_2, o : rv_{\mathtt{i}}(e_1, o_1) \wedge rv_{\mathtt{j}}(e_2, o_2) \wedge o_1 \neq o_2 \wedge rv_{\mathtt{set}}(o_1, o) \wedge rv_{\mathtt{set}}(o_2, o)$
$same_{\mathtt{v},\mathtt{w}}(e_1, e_2)$	$\exists s : rv_{\mathtt{v}}(e_1, s) \wedge rv_{\mathtt{w}}(e_2, s)$

Figure 10: 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 Translation
`x.v = new Set()`	$pt_x(e_1) \vee pt_y(e_2) \rhd same_{\mathtt{v},\mathtt{v}}(e_1, e_2) := (e_1 = e_2)$ $pt_x(e_1) \rhd same_{\mathtt{v},z}(e_1, e_2) := 0 \qquad \forall z \in VF - \{\mathtt{v}\}$ $pt_x(e_2) \rhd same_{z,\mathtt{v}}(e_1, e_2) := 0 \qquad \forall z \in VF - \{\mathtt{v}\}$ $pt_x(e_2) \rhd iterof_{z,\mathtt{v}}(e_1, e_2) := 0 \qquad \forall z \in I$
`x.v.add()`	$(\exists e_2 : pt_{\mathtt{x}}(e_2) \wedge iterof_{\mathtt{i},\mathtt{v}}(e_1, e_2)) \rhd stale_{\mathtt{i}}(e_1) := 1 \qquad \forall \mathtt{i} \in IF$
`x.i.remove()`	$(\exists e_2 : pt_{\mathtt{x}}(e_2) \wedge mutx_{\mathtt{j},\mathtt{i}}(e_1, e_2)) \rhd stale_{\mathtt{j}}(e_1) := 1 \qquad \forall \mathtt{j} \in IF$

Figure 11: Update formulae for the instrumentation predicates of HCMP shown in Fig. 10.

abstractions. Statements in the client program that do not invoke component methods or manipulate component references are, however, translated using a suitable standard abstraction, such as the one in Fig. 9. As explained earlier, the choice of the standard abstraction depends on the kind of heap analysis desired for analyzing the client's heap.

Thus, the specialized component abstraction leads to a certification algorithm that is parametric with respect to the heap analysis used for the *client's heap*. We refer to the parametric certification algorithm obtained for CMP as HCMP.

5.5 Abstract Interpretations For TVP

We now present a brief overview of how the generated TVP program is analyzed using TVLA. TVLA is an abstract interpretation system for TVP based on 3-valued logic, which extends boolean logic by introducing a third value $1/2$ denoting values which may be 1 or 0. A 3-valued structure over a set of predicate variables P is a pair $\langle U, \iota \rangle$ where U is the universe, and ι is a function mapping predicate variables to their values: for every predicate $p \in P$ of arity k, $\iota(p) \colon U^k \to \{0, 1/2, 1\}$.

3-valued structures (as well as 2-valued structures, which are just special cases of 3-valued structures) can be *abstracted* into smaller 3-valued structures by merging multiple individuals into one, and by approximating the predicate values appropriately.

TVLA users can control this abstraction process by identifying a subset A of unary predicates to be the *abstraction predicates*. TVLA's abstraction mechanism merges all individuals having the same value for all abstraction predicates into one individual. The maximum size of the universe in a structure produced by abstraction is $3^{|A|}$. 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, which approximates all structures that may arise at that point. While this distinction was irrelevant for SCMP, it does in principle have a bearing on the precision of the analysis for CMP, since it affects the accuracy of the generic analysis of the *client* heap. Somewhat surprisingly, our empirical results (see Section 7) showed that the relational version of the TVLA certification engine had no precision advantage over the independent attribute version for the benchmark clients we studied. This seems to provide further evidence of the role that specialized component abstractions play in yielding precise results. We refer the reader to [26] for more details.

6. MUTATION-RESTRICTED SPECIFICATIONS

We now show that for the class of *mutation-restricted* specifications, our derivation procedure terminates, producing a finite, precise abstraction for the component. This class includes all the examples presented in Section 2.2. Our result also implies that the results in Section 4 and Section 5 for CMP also apply to all problems in this class. Thus, for problems in this class, precise meet-over-all-paths certification can be done in polynomial time for client programs that do not use heap-based component-references. The results established in this section also imply that the precise meet-over-all-paths solution for certain restricted classes of alias analysis problems is computable in polynomial time, a result that is interesting in its own right. Proofs have been omitted due to space constraints.

Note that our abstraction derivation approach can produce finite precise abstractions even for specifications that are not mutation-restricted. A prominent example is CMP. Finding a better characterization of the class of specifications for which the approach is guaranteed to yield finite precise abstractions is an open problem.

We first introduce some terminology. A component specification is said to be *alias-based* if all its preconditions are alias conditions (i.e., of the form "`requires` $\alpha = \beta$"). A component field `f` is said to be *immutable* if the field `f` of a component is assigned a value only when the component is constructed; otherwise, it is said to be a *mutable* field. A component specification is *mutation-free* if all components in the specification have only immutable fields. (Note that these definitions are with respect to the Easl specification of the component, not any underlying implementation, which may freely use mutable fields.)

Let TG denote a set of component types. Assume that TG is *closed*: i.e., if type T_1 in TG has a field of type (pointer to) T_2, then T_2 is also assumed to be in TG. We define the type graph of TG to be the graph consisting of a node for each type in TG, and an edge $T_1 \to T_2$ labelled f for every field f in type T_1 of type pointer to T_2. Let $||TG||$ denote the number of different paths in the type graph of TG.

Theorem 1. *For any mutation-free, alias-based, straight-line component specification over TG, the component abstraction algorithm will produce a disjunctive abstraction consisting of at most $||TG||^2$ predicate families.*

We note that the above theorem applies to problems IMP and AOP described in Section 2.2. We now consider a class of component specifications that utilize a restricted form of destructive heap update. Destructive heap update statements are statements of the form "$\alpha.f := rhs$". A specification is said to have the restricted heap update property if the right-hand side of every destructive heap update statement in the specification is a newly constructed object (or a "tree" of newly constructed objects, where each object has pointers only to other newly constructed objects).

A specification with restricted heap update is said to have mutation depth k if there are at most k mutable fields in any path in the type graph of the specification.

Theorem 2. *For any alias-based, straight-line component specification over TG of mutation depth 1, the component abstraction algorithm will produce a disjunctive abstraction consisting of at most $||TG||^4$ predicate families.*

We note that the above theorem applies to problem GRP described in Section 2.2.

The above results focus on the component abstraction problem. However, these results also imply the following upper bounds for certain alias analysis problems, if the concept of mutation depth is generalized to whole programs in the obvious way. These results qualify the well known results that alias queries of depth 2 are hard [23, 27].

Theorem 3. *(a) Must-alias and may-alias queries of mutation depth 0 can be precisely answered in polynomial time (treating the type graph size as fixed). (b) Must-alias queries of mutation depth 1 can be precisely answered in polynomial time for programs with restricted heap update (treating the type graph size as fixed).*

We now present some lower bounds. These are adaptations of well-known intractability results of alias analysis [27] 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 [27].

Theorem 4. *(a) May-alias and must-alias queries of mutation depth 1 are PSPACE-hard. (b) May-alias queries of mutation depth 1 is PSPACE-hard even for programs with restricted heap update. (c) Must-alias queries of mutation depth 2 is PSPACE-hard even for programs with restricted heap update.*

7. EMPIRICAL RESULTS

We have prototyped several variants of the HCMP algorithm using Soot [36] and TVLA [26]. We wished both to evaluate the precision of the algorithm (i.e., the number of false alarms produced), and to understand the cost/precision tradeoffs in the analysis design space.

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 existing techniques [40, 31].

7.1 Engineering Aspects

Our Soot-based translator from Java to TVP uses liveness information to reduce the number of predicates required for the analysis and to restrict their scope. We take advantage of Soot's implementation of *Class Hierarchy Analysis* [12] to conservatively construct a method call graph. In addition, Soot treats exceptions by constructing a control flow edge from each statement that may throw an exception to all potential corresponding handlers.

7.2 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:

(i) Using a relational versus independent attribute approach for modelling TVLA structures at every program point.

(ii) Context-sensitive versus context-insensitive treatment of client method calls.

(iii) Using allocation sites versus variables names to distinguish client heap cells. The variable names approach [37, 33] merges two heap nodes if the set of variables and fields pointed to them are the same. The allocation site approach merges two nodes if they are allocated at the same allocation site in the client program [6, 21].

Our empirical observations were as follows:

(i) The independent attribute approach yielded an implementation that was *as precise* as the relational one, and faster. We believe that the precision of the independent attribute approach is due to the disjunctive abstractions we use as instrumentation predicates.

(ii) The context-sensitive algorithms performed better than the corresponding context-insensitive ones with respect to *both efficiency and precision*.

(iii) The variables names approach yielded more precise results than the allocation site approach.

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

7.3 Results

Our experimental results indicate that our analysis is quite fast and precise, producing only one false alarm over the test suite. (The false alarm was produced by a conservative modelling of calls to Java libraries outside the scope of the analysis.) Fig. 12 displays the results of the experiments. The test programs, which use JCF intensively, are available at [38]. The `KernelSuite`, designed to "stress test" the analysis, includes numerous examples illustrating various difficult aspects of CMP. `MapTest` is from [22]. `IteratorTest` and `MapDemo` are examples from [39]. `JFE` is our own implementation's front-end. The experiments were performed on a machine with a 1 Ghz Pentium 4 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 50 Mb.)

7.4 Future Enhancements

We are currently improving our implementation to allow analysis of significantly larger programs. The improvements from which we expect to derive the most benefits are (i) slicing away portions of the program irrelevant to the component, and (ii) improving the representation of first-order structures.

8. INTERPROCEDURAL SCMP

The class of certifiers described in Section 4 yields precise meet-over-all-paths solutions for *single-method* clients in polynomial time. The class of TVLA-based certifiers described in Section 5 handles *interprocedural* CMP clients, (i.e., which contain calls among multiple client methods); however, the resulting solution will not be precise in general. In this section, we show that the SCMP certifier of Section 4 can be extended to compute the precise *interprocedural*

Benchmark	# Classes	# Methods	# Lines of Code	# CFG Nodes	# Errors Reported	# False Alarms	Analysis Time (s)	Analysis Space (Mb)	# TVLA Structures
KernelSuite	5	27	683	2150	15	0	60.09	18.66	4363
MapTest	2	9	335	424	1	0	61.20	19.87	4937
IteratorTest	3	10	126	154	0	0	0.23	4.18	208
MapDemo	1	3	33	32	0	0	0.01	1.13	26
JFE	1	45	2396	2896	1	1	236.34	49.10	9878

Figure 12: An empirical evaluation of HCMP.

meet-over-all-valid-paths solution for multi-method SCMP clients in polynomial time. Unlike the techniques described in Sections 4 and 5, the methods in this section pertain only to SCMP; deriving precise interprocedural analyses for more general classes of component specifications is an open problem.

In the absence of recursive procedures with local variables, previous techniques [30] can be used to solve interprocedural SCMP. Recursive procedures with local variables, however, complicate issues as illustrated by the following example:

```
/*   */ 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 cannot 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 change). This complicates matters since the number of *instances* of local variables of recursive procedures is unbounded.

We now show how we can handle these issues. We assume, for now, that no global variables are used. For any procedure Q, let Formals(Q) denote the set of formal parameters of procedure Q of type Set or Iterator. Define MayMod(Q) to be the subset of $Formals(\mathtt{Q})$ such that: a collection variable S is in MayMod(Q) iff the execution of Q may modify the collection S and an iterator variable I is in MayMod(Q) iff the execution of Q may modify the collection underlying the iterator I via iterator I, assuming 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 [3] and can be computed very efficiently [8].

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 (...) AC₁.add(""); ... ; if (...) ACᵢ.add("");
if (...) AI₁.remove(); ... ; if (...) AIⱼ.remove();
```

Standard techniques can be used to analyze the whole program using the above method for analyzing a single procedure. We refer the reader to [29] for more details.

Interaction between local and global variables: In the presence of both local and global variables, the information MayMod(Q) is not sufficient to handle calls to procedure Q. The problem is that a procedure call may modify the value of a global variable, say g. This can have the effect of changing the value of an instrumentation predicate such as $mutx_{\mathtt{g},\mathtt{l}}$, where l is a local variable of the calling procedure. Modelling a procedure call to account for such effects requires summary information similar to that required for precise interprocedural slicing, which can be computed efficiently [30]. We refer the reader to [29] for more details.

9. RELATED WORK

In addition to the verification techniques based on predicate abstraction discussed in Section 4.5, several other research efforts have goals that are similar, to varying degrees, to ours.

Abstraction without predicate discovery Bandera [9] differs from our work in that it does not use predicate discovery. Instead, it relies on program slicing and user-provided abstractions to abstract programs into finite state models.

Verification without abstraction ESC-Java [25] relies on theorem-proving and program invariants to do verification. Traditionally, users provided the invariants, but recent work has focused on automatically "discovering" invariants [15] by generating candidate invariants heuristically, then eliminating those that are not irrefutable. However, ESC-Java's overall approach to verification is quite different from our approach. Also, unlike ESC-Java, our approach is *conservative*.

Interprocedural Shape Analysis Noam Rinetzky [32] proposes improving the efficiency of interprocedural shape analysis of programs manipulating abstract data types by defining a special, hand-crafted semantics for a linked-list abstract data type.

Type Checking Other approaches to improved static checking includes languages with advanced type systems (e.g., [13]). We are not aware of any type system that can check for the kind of properties we have looked at in this paper.

Incomplete Concrete State-Space Exploration Jackson and Fekete [19] address the Concurrent Modification Problem by *partially* exploring the concrete state space of the program [20]. While this approach never produces false alarms, it is inherently incomplete.

We refer the reader to [29] for more discussion on related work.

Acknowledgments

We thank Sean McDirmid for introducing us to the Concurrent Modification Problem, the SOOT team and Roman Manevich for the assistance they provided with the implementation, and Robert O'Callahan, Nurit Dor, Ran Shaham, Carlos Varela, Reinhard Wilhelm, and Eran Yahav for their helpful feedback on the paper.

10. REFERENCES

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

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

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

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

[5] 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.

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

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. Computer Aided Verification*, pages 154–169, 2000.

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

[9] 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*, pages 439–448, June 2000.

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

[11] P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[12] 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.

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

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

[15] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. Technical Report 2000-003, Compaq Systems Research Center, 2000.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

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

[18] 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)*, pages 72–83, Haifa, Israel, June 1997.

[19] D. Jackson and A. Fekete. Lightweight analysis of object interactions. In *Proc. Intl. Symp. on Theoretical Aspects of Computer Software*, Sendai, Japan, October 2001.

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

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

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

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

[24] G. T. Leavens. The Java Modeling language (JML). http://www.cs.iastate.edu/∼ leavens/JML.html.

[25] 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.

[26] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In J. Palsberg, editor, *Proc. Static Analysis Symp.*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer-Verlag, 2000.

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

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

[29] G. Ramalingam, A. Warshavsky, J. Field, and M. Sagiv. Deriving specialized heap analyses for verifying component-client conformance. Technical Report RC22145, IBM T.J. Watson Research Center, 2001.

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

[31] N. Rinetskey and M. Sagiv. Interprocedural shape analysis for recursive programs. In R. Wilhelm, editor, *Proc. Intl. Conf. on Compiler Construction*, volume 2027 of *LNCS*, pages 133–149. Springer-Verlag, 2001.

[32] N. Rinetzky. Interprocedural shape analysis. Master's thesis, Technion-Israel Institute of Technology, Haifa, Israel, Dec. 2000.

[33] 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.

[34] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 105–118, 1999.

[35] H. Saïdi. Model checking guided abstraction and analysis. In J. Palsberg, editor, *Proc. Static Analysis Symp.*, volume 1824 of *Lecture Notes in Computer Science*, pages 377–389. Springer-Verlag, 2000.

[36] 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 *Proc. Intl. Conf. on Compiler Construction*, pages 18–34, Mar. 2000.

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

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

[39] M. A. Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, second edition, 2001.

[40] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 27–40, 2001.