Master Thesis

Deriving Specialized Heap Analyses for Verifying Component-Client Conformance

Alex Warshavsky
Tel-Aviv University

September 2001

Abstract

We are concerned with the problem of statically *certifying* (verifying) whether the client of a software component or library conforms to the component's constraints for correct usage. We focus here on constraints that involve relationships among dynamically-allocated objects in a potentially unbounded heap. A number of natural component conformance constraints not expressible in propositional logics or as finite state machines are expressible as those constraints, e.g., the *Concurrent Modification Problem* (CMP), which arises when certain classes defined by the Java Collections Framework are misused. Using CMP as a running example, we show how conformance certification can be efficiently carried out in a *staged* fashion for certain classes of specifications. In the first stage, we derive a component-specific set of abstractions for use in analysis of arbitrary clients. Unlike verification approaches that analyze a specification and client code together, our technique can take advantage of computationally-intensive symbolic techniques without affecting the performance of client analysis. In the second stage, the generated abstractions are 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. We describe several different classes of certifiers with varying time/space/precision tradeoffs, all of which use the same abstractions. Of particular note are certifiers that run in *polynomial* time for a restricted, but still quite useful class of client programs; other verification approaches would require exponential time for this class. 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.

ACKNOWLEDGMENTS

This thesis was created under the supervision of Dr. Shmuel Sagiv with close cooperation of G.Ramalingam, John Field and Deepak Goyal from IBM T.J. Watson Research Center.

I would like to thank:

- Dr. Shmuel Sagiv, without whom this thesis would never be written.
- G. Ramalingam, John Field and Deepak Goyal, who contributed enormously to this work.
- Katia for here love and endless encouragements.
- My family for their care and support.
- Litovky-Orlov family for their care and useful discussions.
- Eran Yahav, Noam Rinetzky, Ran Shacham, Guy Laden, Nurrit Dor and Sean McDirmid for their reviews and discussions.
- Roman Manevich and the SOOT team at McGill university for the assistance provided with the implementation of the certifier.
- The IBM Haifa Research Center and Israeli Academy of Science for their financial support.

Contents

ACKNOWLEDGMENTS **ii**

1 Introduction **1**
- 1.1 The Concurrent Modification Problem 2
- 1.2 Our Approach to Conformance Certification 3
- 1.3 Results 5

2 Our Certification Approach and the Concurrent Modification Problem **8**
- 2.1 The Specification Language EASL 9
- 2.2 Specifying CMP 9
- 2.3 A Running Example 12

3 Applying Standard Heap Analyses to CMP **13**
- 3.1 A TVP Semantics for EASL 13
- 3.2 Generating An Abstract Interpreter From the TVP Semantics . . 16
- 3.3 The Effectiveness of Standard Heap Analyses 18

4 Systematic Generation of CMP-Specific Heap Analysis **19**
- 4.1 The Derivation Procedure 19
- 4.2 Application to SCMP 20
- 4.3 The Complexity of the SCMP Algorithm 22
- 4.4 Extending SCMP to CMP 24

5 A Generic Front End for Java **26**
- 5.1 Specification files 26
- 5.2 Implementation Details 27

6 Empirical Results **28**
- 6.1 Engineering Aspects of the Implementation 28
- 6.2 Analysis Design Tradeoffs 29
- 6.3 Benchmarks 29

7 Related Work and Conclusions **35**

8 Further Work **39**

A Front End User Manual **46**
A.1 Functionality 46
A.2 Running example 47
A.3 The Specification File 50
A.4 Translation Rules 51

List of Tables

3.1 A TVP semantics for the pointer manipulation statements of Easl. 16
3.2 A TVP semantics for calls to `Set` and `Iterator` methods. . . . 17

4.1 The nullary instrumentation predicates used by SCMP. 21
4.2 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 4.1 can be updated. 22
4.3 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. 24
4.4 Update formulae for the instrumentation predicates of HCMP shown in Table 4.3. 25

6.1 The analyzed programs. Kernel benchmarks and a `VCMPKiller` examples are artificial programs created to illustrate interesting cases of JCC uses. `MapTest` is from [39]. `IteratorTest` and `MapDemo` are examples from [66]. `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. . . . 30
6.2 The benchmark results of the analyzed programs for the context-sensitive, variables-names based, independent attribute variant of the algorithm. 31
6.3 The benchmark results of the analyzed programs for the context-sensitive, variables-names based, relational variant of the algorithm. 31
6.4 The benchmark results of the analyzed programs for the context-sensitive, allocation-site based, independent attribute variant of the algorithm. 32
6.5 The benchmark results of the analyzed programs for the context-sensitive, allocation-site based, relational variant of the algorithm. 32

6.6 The benchmark results of the analyzed programs for the context-insensitive, variables-names based, independent attribute variant of the algorithm. 33
6.7 The benchmark results of the analyzed programs for the context-insensitive, variables-names based, relational variant of the algorithm. 33
6.8 The benchmark results of the analyzed programs for the context-insensitive, allocation-site based, independent attribute variant of the algorithm. 34
6.9 The benchmark results of the analyzed programs for the context-insensitive, variables-names based, relational variant of the algorithm. 34

List of Figures

1.1 An erroneous Java program fragment throwing `CME`. 4

2.1 The EASL language. X^* denotes zero or more occurrences of X, while $X^{\circledast}$ denotes a comma-separated list of zero or more occurrences of X. 10
2.2 An EASL specification of CMP. 11
2.3 A Java fragment illustrating CMP 12

3.1 A 2-valued structure representing the concrete program state at label 6. 15
3.2 A 3-valued structure (abstract state) that approximates the 2-valued structure (concrete state) shown in Figure 3.1. 17

4.1 An illustration of the abstract state maintained by SCMP. . . . 23
4.2 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. 23

A.1 Example Java progam, serves as a running example 48
A.2 The Jimple representation of the program from Figure A.1 49
A.3 The specification file for the running example program shown in Figure A.1. 50
A.4 Set declarations in the rule file for the running example program shown in Figure A.1. 50
A.5 The assignment and control flow Jimple statements in the rule file. 51
A.6 The method invocation Jimple statements in the rule file. 52
A.7 A TVP program generated from the Jimple program shown in Figure A.2. Only the modeled statements are shown. 53
A.8 Sets for a TVP program shown in Figure A.7 54
A.9 The syntax of a specification file. 54
A.10 The syntax of the translation rules. The <assignment> rules are shown in Figure A.11. 55
A.11 The <assignment> syntax rules. 56

Chapter 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. This thesis aims to help ease the use of software components [49, 61] 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 thesis, we consider a "component" to be any *object-oriented software library*, and focus on components written in Java [25].

This thesis concentrates on interesting, non-trivial problems whose verification requires reasoning about relationships among dynamically-allocated objects in a potentially unbounded heap. The verification is carried out using TVLA [42], a system for generating abstract interpreters from a TVP specification (a language based on first order logic for specifying operational semantics).

TVLA system was used to solve a number of interesting problems [55]. However, the experimental results were performed mostly on relatively small benchmarks, which makes it hard to verify the scalability of the analysis. One of the difficulties for such verification, is the need to transform benchmark programs into TVP specification, which prior to this thesis were carried out by hand.

As part of this thesis, I have developed a generic front end. The front end generates a set of TVP files from a given Java program automatically. With the help of the front end we succeded to perform the analysis on medium programs and hope to scale our analysis even further in the future.

1.1 The Concurrent Modification Problem

Java 1.2 has eased the development of programs with introduction of Java Collections Framework (JCF) [7], which is commonly used by Java developers in order to avoid the need to reimplement complex data structures.

In this thesis, we will extensively study a problem that is a microcosm of our broader goals. The problem, which we will call the *Concurrent Modification Problem* (CMP), arises in the context of the JCF.

JCF `Iterator`s are used to iterate over the contents of an underlying collection (e.g., a `HashSet`, which implements the `Set` interface). It is not obvious what should be the state of an iterator if while iterating over a data type, the data type has been structurely modified. JCF has implemented, what they call as, *fail-fast* iterators, i.e., whenever the data type is structurely modified while the iterator is iteratating over it, the iterator raises a `ConcurrentModificationException` (or `CME`). Formally speaking, 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* (at runtime), and throw 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.

Whoever uses JCF classes knows how easy it is to raise a CME. In many cases the reason for CME is obvious and therefore easily fixed while debugging the application. It is easy to find the reason for the exception when the iterations and the updates of the underlying data type are closed in the source code. Nevertheless, the update may be far away from the iteration loop, nested deep in a call stack and occur only in irregular situations. In those cases CME's are not easy to find, and they may manifest themselves long after the application deployment.

Consider the Java code fragment in Figure 1.1. 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. This code fragment is an abstraction of situations where `CME` was thrown in the code for the Soot [62] framework and the TVLA [42] system (both used to implement the analyses we present in the sequel). We have found that `CME` is often a manifestation 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. Simple search on the internet revealed that CME is quite widespread in practice as well. It manifest itself in Java Virtual Machines (Kaffe framework [39]), Java Compilers (JLTOOLS [36]), Java Application Servers (JOnAS [50]) and other publically available tools [3, 4].

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.

Our approach to static component certification is based on *abstract interpretation* [12] of the client program, which ensures that every potential violation of the component's conformance constraints is detected. The absence of any detected errors thereby guarantees that a client contains no violations. However, a potential drawback of abstract interpretation 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 thesis, 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:

Conformance specification via heap properties

First, the component designer or implementer writes a conformance specification. Such a specification will typically describe properties of the component that the designer regards as critical to correct client usage.

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 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, 44]. In particular, heap properties can be used to model the interactions among a potentially *unbounded* number of component objects, a prerequisite for analyzing CMP.

For example, CMP cannot be specified using a finite state machine when the client code is not available. Furthermore, in the presence of unbounded number of collection objects, this cannot be done even when the whole program is available.

In the sequel, we will use *abstract Java programs* (written in an experimental specification language called EASL[1]) for specifying the conformance constraints

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

```
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 1.1: An erroneous Java program fragment throwing CME.

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 specification languages such as JML [40] can also express many of the properties with which we are concerned.

Off-line generation of component-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., [37, 47, 27, 58, 57]). We investigated the applicability of several heap analysis algorithms to CMP, including a powerful *relational* [48, p. 248] heap analysis implemented using the TVLA system [42]. 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 the component's conformance specification is used to derive a set of component-specific abstractions. The generated abstractions are used to sharpen, or *instrument* [48, p. 131] the representation of the program state relevant to the component. 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.

Performing the analyses

The generated abstractions are then incorporated into an abstract-interpretation based static analysis engine to yield a *certifier*. The engine performs a fixpoint analysis of a client program using the abstractions generated in the previous phase. Different analysis engines may incorporate varying time/space/precision tradeoffs.

Finally, client programs are fed to the certifier to conservatively determine whether possible violations of the conformance constraints can occur.

1.3 Results

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

In Section 2, we show how CMP can be naturally specified as *abstract programs* in EASL.

In Section 3.1, we show how a "vanilla" form of conformance certification can be carried out using classical abstract interpretation techniques. We will represent abstract states as *first-order structures*, and use the TVLA abstract interpretation system [42] as a configurable analysis engine. This combination allows the vanilla certifier to be generated directly from a simple translation of the EASL specification and the client program. For CMP, the vanilla analysis can be viewed as a form of high-precision generic heap analysis. However, in Section 3.3, we show that that the results of the analysis are too imprecise to be useful as a practical certifier for CMP and similar problems.

In Section 4, we show how to remedy the imprecision of the vanilla appproach by first generating component-specific *disjunctive abstractions*. The abstraction derivation process combines the notions of *weakest precondition* [17] and *disjunctive completion* [13, 24] to yield a set of first-order formulae that characterize the properties needed to perform precise (in the sense of meet-over-all-paths) intraprocedural analysis of any client program.

Since the abstraction generation process is carried out *offline*, prior to analyzing a particular client, it can take advantage of manual or computationally-intensive symbolic techniques that would be impractical in online verification approaches. Formally, the disjunctive abstractions we derive become *instrumentation predicates* for the TVLA system. In previous analysis algorithms developed using TVLA, the instrumentation predicates, which are critical to the performance and precision of the analysis, were developed in an *ad hoc* manner. Our work here may be viewed as a step towards deriving instrumentation predicates (and, more generally, instrumented semantics for abstract interpretations) systematically.

We show in Section 4.2 that for Java programs in which references to collections and iterators are stored only in local or static variables (rather than in object fields), we can use the disjunctive abstractions derived for CMP to yield a precise, polynomial-time intraprocedural analysis. We will refer to this restricted version 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 and *EXPTIME*-hard in the interprocedural case [46].

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.

Section 4.4 shows how to handle the *unrestricted* CMP problem by adapting the CMP-specific distributive abstractions to serve as *instrumentation* for a generic analysis of the client heap. The extended CMP algorithm we present is *adaptive*, in the sense that it yields precise results in polynomial time when the client program is in the SCMP class.

Section 5 gives an insight into main ideas and implementation details of the generic front end, used to translate client Java programs into the TVP specifications. Appendix A contains a user-manual for the front end.

In Section 6, 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 a number of different 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.

A detailed comparison to related work is given in Section 7. Finally, Section 8 describes further work.

Chapter 2

Our Certification Approach and the Concurrent Modification Problem

Our 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[1] is fed to the certifier, and a report is produced indicating whether possible violations of the conformance constraints have occurred.

[1]In the sequel, we will assume there is only a single component and a single client; however, our approach is intended to accommodate clients that use multiple components, or components that incorporate other components.

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

- Phase 1 uses a specification language called EASL, described in more detail in Section 2.1.
- In Phase 2, the EASL specification is translated to the *TVP* language (see Section 3) used as input to the TVLA [42] program analysis system. In Section 3, an EASL 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 [62] 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 6.

2.1 The Specification Language EASL

EASL specifications take the form of *abstract* Java programs (similar facilities are found in the JML specification language [40]). The syntax of EASL is depicted in Figure 2.1. EASL 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 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.

While a discussion of the merits of an "imperative" specification language is beyond the scope of this thesis, we note that EASL is similar in spirit to several existing "wide spectrum languages" that combine specification statements with conventional programming constructs [45, 30]. 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.

2.2 Specifying CMP

We now show how CMP can be specified in EASL 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

$$
\begin{array}{rcl}
\langle\text{Component}\rangle & \rightarrow & \texttt{class}\ \langle\text{Id}\rangle\ \{\ \langle\text{Field}\rangle^{*}\ \langle\text{Method}\rangle^{*}\ \} \\
\langle\text{Field}\rangle & \rightarrow & \langle\text{Type}\rangle\ \langle\text{Id}\rangle; \\
\langle\text{Type}\rangle & \rightarrow & \text{Boolean} \mid \langle\text{Id}\rangle \\
\langle\text{Method}\rangle & \rightarrow & \langle\text{Type}\rangle\ \langle\text{Id}\rangle\ (\langle\text{Arg}\rangle^{\circledast})\ \{\ \langle\text{Stmt}\rangle^{*}\ \} \\
\langle\text{Arg}\rangle & \rightarrow & \langle\text{Type}\rangle\ \langle\text{Id}\rangle \\
\langle\text{Stmt}\rangle & \rightarrow & \texttt{requires}\ \langle\text{Cond}\rangle; \\
 & \mid & \langle\text{Lhs}\rangle := \langle\text{Exp}\rangle; \\
 & \mid & \{\ \langle\text{Stmt}\rangle^{*}\ \} \\
 & \mid & \texttt{while}\ \langle\text{Cond}\rangle\ \texttt{do}\ \langle\text{Stmt}\rangle; \\
 & \mid & \texttt{if}\ \langle\text{Cond}\rangle\ \langle\text{Stmt}\rangle\ [\ \texttt{else}\ \langle\text{Stmt}\rangle\ ]\ ; \\
 & \mid & \texttt{return}\ \langle\text{Exp}\rangle; \\
\langle\text{Lhs}\rangle & \rightarrow & \texttt{this} \mid \langle\text{Id}\rangle \mid \langle\text{Lhs}\rangle.\langle\text{Id}\rangle \\
\langle\text{Exp}\rangle & \rightarrow & \texttt{null} \mid \texttt{new}\ \langle\text{Id}\rangle\ (Exp^{\circledast}) \\
 & \mid & \langle\text{Lhs}\rangle \mid \langle\text{Cond}\rangle \\
\langle\text{Cond}\rangle & \rightarrow & \texttt{true} \mid \texttt{false} \mid \langle\text{Lhs}\rangle \\
 & \mid & \langle\text{Exp}\rangle == \langle\text{Exp}\rangle \\
 & \mid & !\langle\text{Cond}\rangle \\
 & \mid & \langle\text{Cond}\rangle\ \texttt{and}\ \langle\text{Cond}\rangle \\
 & \mid & \langle\text{Cond}\rangle\ \texttt{or}\ \langle\text{Cond}\rangle
\end{array}
$$

Figure 2.1: The EASL language. X^{*} denotes zero or more occurrences of X, while $X^{\circledast}$ denotes a comma-separated list of zero or more occurrences of X.

```
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 2.2: An EASL specification of CMP.

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. 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, indicating that that particular iterator may continue to be used safely.

Figure 2.2 contains an EASL 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 2.2 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 2.3 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

```
/* 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 2.3: A Java fragment illustrating CMP

`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.

Chapter 3

Applying Standard Heap Analyses to CMP

In this section, we show how, given an EASL specification of a component, we can certify clients of the component using standard "off the shelf" heap analyses, using CMP to illustrate the approach. Our certification process is based on abstract interpretation over first-order structures. 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 [42]. We present a simple TVP semantics for EASL. This semantics, in conjunction with TVLA, yields an algorithm for CMP that we will refer to as VCMP (for "vanilla" CMP), since it uses abstractions that are independent of the component specification. Readers not interested in the formal details may skip to Section 3.3.

3.1 A TVP Semantics for EASL

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 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 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.

Procedures are modelled 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 $rv[\texttt{localvar}](\texttt{o}_1, \texttt{o}_2)$ is true iff the variable `localvar` of activation record o_1 refers to object o_2. We refer the reader to [53] for additional details about how procedures and local variables may be modelled using TVLA system.

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 thesis 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 3.1 depicts a 2-valued structure representing the state of the program of Figure 2.3 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 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

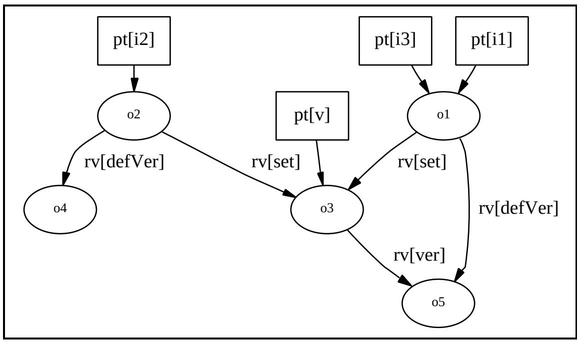

Figure 3.1: A 2-valued structure representing the concrete program state at label 6.

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 $guard \rhd lhs := rhs$ 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 3.1 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 3.2 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 3.1 to the method specifications shown in Figure 2.2, 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 3.2 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

EASL Condition	TVP Precondition
`x == y`	$\forall o : pt[\mathtt{x}](o) \Leftrightarrow pt[\mathtt{y}](o)$
EASL 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 3.1: A TVP semantics for the pointer manipulation statements of EASL.

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 3.1.

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

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.

Figure 3.2 illustrates this abstraction process. The figure depicts a 3-valued structure obtained from the 2-valued structure shown in Figure 3.1 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.

TVLA users can control this abstraction process by identifying a subset A of unary predicates to be the *abstraction predicates*. TVLA's abstraction mecha-

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

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

Table 3.2: A TVP semantics for calls to `Set` and `Iterator` methods.

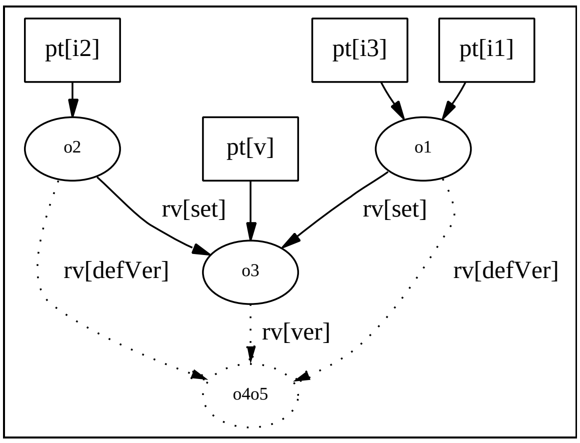

Figure 3.2: A 3-valued structure (abstract state) that approximates the 2-valued structure (concrete state) shown in Figure 3.1.

nism merges all individuals having the same value for all abstraction predicates into one abstract individual. The maximum size of the universe in a structure produced by abstraction is $3^{|A|}$. Hence, the number of different abstract structures that can arise at a program point is bounded. 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. Finally, TVLA provides a mechanism to check if structures arising at a program point satisfy specific properties, which we use to do the conformance checking.

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

Consider the abstract interpretation obtained by treating just predicates of the form *pt*[`var`] as abstraction predicates, as used in VCMP analysis. (This corresponds to a heap analysis that merges nodes in a "storage shape graph" together iff they are pointed to by the same set of variables [64, 56], and its worst-case complexity is doubly exponential in the number of pointer variables.)

It is straightforward [57] to use various other alternative semantics to generate alternative heap analyses, such as [38, 8] or [37]. 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 3.2 *represents* the concrete program state shown in Figure 3.1. 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.

Chapter 4

Systematic Generation of CMP-Specific Heap Analysis

Vanilla certifiers are ineffective, in part, because they use fixed abstractions that are *independent* of the certification problem. Note that a component's state can be changed (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 abstraction* (of the components' state), which can be used as the basis for more effective certification algorithms. In this section, we present a procedure for deriving such specialized abstractions, and illustrate it by applying it to CMP. We will initially make the simplifying assumption that there are no component references in the heap, and later show how component references in the heap can be handled. We will also show that the abstraction identified yields a polynomial time certifier for a class of Java programs (which we refer to as SCMP) in which references to collections and iterators are stored only in local or static variables (rather than in object fields).

4.1 The Derivation Procedure

The goals of this derivation procedure are to (a) identify the information about a component's state that is relevant to the certification process (which we will refer to as instrumentation predicates [57]) and (b) determine how this relevant state information is changed by any call to a component method.

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`, $WP(\texttt{S}, \varphi)$, is a *candidate instrumentation formula*. ($WP(\texttt{S}, \varphi)$ is a formula that holds before the execution of `S` iff φ holds after the execution of `S`.)

The specific form of rule 2 is intended to enable the use of efficient independent attribute analysis without losing the precision of relational analysis. 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. 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 an imprecision (in the case of independent attribute analysis). This is related to the notion of disjunctive completion [13, 24] and distributivity in static analysis.

These rules yield not only instrumentation predicates, but also their update formulae. Let φ_0 be a candidate instrumentation predicate, and let the weakest precondition of φ with respect to component method `M` be $\varphi_1 \vee \ldots \vee \varphi_k$. Let p_i denote the instrumentation predicate corresponding to candidate φ_i. Then, the abstraction of method `M` includes the precise update formula "$p_0 := p_1 \vee \ldots \vee p_k$".

In Section 4.2, we show that the derivation process above converges quickly for CMP. Although, the derivation process can be applied to arbitrary EASL specifications, it 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 specifications is an interesting open problem. 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.

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.

4.2 Application to SCMP

We now illustrate this procedure for SCMP. For the sake of readability, we will express formulae using EASL 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`.

Predicate	Meaning (in EASL)	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 4.1: The nullary instrumentation predicates used by SCMP.

Step 2: The next step is to consider how the execution of different `Set` and `Iterator` methods will affect the value of the predicate $invalid[\mathtt{i}]$. 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 ($invalid[\mathtt{i}]$ 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 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

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

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!

EASL **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 4.2: 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 4.1 can be updated.

Table 4.1 presents a formal definition of the instrumentation predicates using core predicates and explains their intended meaning in EASL. The update formulae for the instrumentation predicates is shown in Table 4.2. 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.

Figures 4.2 and 3.2 illustrate the abstract state maintained by the VCMP algorithm described earlier just before and after the statement `i1.remove()` in the program of Figure 2.3, leading to imprecision. Figure 4.1 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.

4.3 The Complexity of the SCMP Algorithm

The specialized semantics in Table 4.2 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, in fact, a member of the finite distributive subset (FDS) framework [52] 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).

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

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

Figure 4.1: An illustration of the abstract state maintained by SCMP.

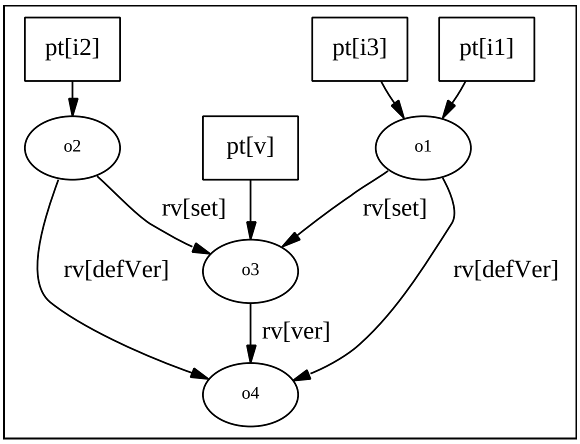

Figure 4.2: An illustration of the abstract state at program point 5 (*i.e.*, just before the execution of the statement labelled 5), as computed by VCMP.

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

Table 4.3: 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.

4.4 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 4.3. These instrumentation predicates generalize the instrumentation predicates of SCMP (see Table 4.1) in a very natural way. For instance, recall that SCMP uses a *nullary* predicate $invalid[\mathtt{i}]$ 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 $invalid_h[\mathtt{i}](o)$ 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 4.2. Table 4.4 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 4.4 which utilize "points-to" information, as represented by predicates of the form $pt[\mathtt{x}]$, for the client's heap. By providing appropriate update formulae for these predicates, ([57] shows how different update formulae can be used to achieve several well known heap analysis techniques) we can generate particular

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

Table 4.4: Update formulae for the instrumentation predicates of HCMP shown in Table 4.3.

instantiations of HCMP.

Our algorithm is also interprocedural, and uses the techniques outlined in [53] to deal with procedures.

Chapter 5

A Generic Front End for Java

The TVLA system is very powerful for performing various kinds of static analyses. It was already used to solve a number of interesting problems [55]. However, the experimental results were performed mostly on relatively small benchmarks, which makes it hard to verify the scalability of the analysis. One of the difficulties for such verification, is the need to transform benchmark programs into TVP specification, which prior to this thesis, were carried out by hand.

As part of this thesis I have developed a generic front end, that translates a Java program into a TVP file automatically. The TVP file is fed into the TVLA system, which performs the analysis. The front end allows to specify the treatment of Java statements and what classes and methods to analyze. The intention is that the resultant analysis conservatively handles situations in which parts of the code are not analyzed. Notice that this may lead to overly conservative and overly expensive analysis. Therefore, the designer of the analysis need to carefully select which classes need to be handled and how.

5.1 Specification files

To perform the translation, front end uses specification files. These specification files describe the classes and the methods to analyze and supply translation rules, which in turn, describe how to translate program statements. As a preprocessor step, the front end translates the Java program into an equivalent Jimple program[1]. Thus, the specification actually manipulates Jimple statements. Readers, not familiar with the Jimple language, can view Jimple statements as simplified Java statements.

Each Jimple statement of the given program may be either ignored by the front end or translated to a local control flow graph; where nodes denote labels

[1]Jimple is a simplification of the Java bytecode [62]

and edges are annotated with TVP actions. Local control flow graphs are later pasted together to form a TVP file.

In addition, the front end can also produce different sets that are used to control the set of generated predicates in the generated TVP files. These sets can also be used to parameterize the predicate-update formulae and other features of the generated analysis.

The rule in the specification file has the following form:

```
<jimple statement> [where <semantic condition>] => <translation>
```

Jimple statements in the specification files are called *modeled statements*. Every *modeled statement* has three parts: a syntax of a Jimple statement, a boolean semantic condition after the **where** keyword and the associated translation. We also allow "abstract" Jimple statements such as assignments to unknown values which may be used to conservatively analyze partial Jimple programs.

We say that a rule *matches* an (abstract) Jimple statement if the syntax of the rule corresponds to the syntax of the statement and the semantic condition holds. In this case, the front end emits the associated translation after replacing the arguments with the actual values. When more than one rule matches, only the first rule in the rule file is applied.

5.2 Implementation Details

The front end is built in pure Java using the Soot [62] framework. Soot parses the Java bytecode and passes the syntax tree to the front end, which in turn, uses the syntax tree for matching rules from the specification files and saves associated translations in memory. When the parsing completes, front end passes through the translations again, performs appropriate backpatching and emits the translation to the TVP file.

A number of Soot optimizations are used to increase the performance of the generated analysis, those optimizations are described in Section 6

For more details see Appendix A.

Chapter 6

Empirical Results

We have prototyped several variants of the HCMP algorithm using the Soot [62] and TVLA [42] 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 [67, 53].

6.1 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 [5].

We take advantage of Soot's implementation of *Class Hierarchy Analysis* [14] 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.

6.2 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:

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 [64, 56] 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 [8, 38].

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.

6.3 Benchmarks

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

Table 6.1 describes the benchmark Java programs; they are publicly available at [65]. 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.

All the experiments were performed on a machine with a 1.4 Ghz Pentium 4 processor, 256 MB 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 77 MB.).

The results for the context-sensitive, variables-names based, independent attribute variant of the algorithm are presented on the Table 6.2. This algorithm is the best for the given set of benchmarks. **Time** is measured in seconds,

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 list cells
Kernel4	166	1103	10	3	Accessing the cells indirectly through auxiliary method calls
Kernel5	55	129	10	2	Figure 1.1
Kernel6	35	32	3	1	Figure 2.3
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 6.1: The analyzed programs. Kernel benchmarks and a `VCMPKiller` examples are artificial programs created to illustrate interesting cases of JCC uses. `MapTest` is from [39]. `IteratorTest` and `MapDemo` are examples from [66]. `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.

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. Table 6.3, Table 6.4, Table 6.5, Table 6.6, Table 6.7, Table 6.8 and Table 6.9 present results of all other algorithms. The `n/a` value in the context insensitive analyses means that the valus is non availabe since the analysis ran out of memory.

We found only one false alarm. It was produced by a conservative modelling of calls to Java libraries outside the scope of the analysis.

Benchmark	**CPU Time (sec)**	**Space (Mb)**	**Struct**	**ERR**	**FA**
Kernel1	0.19	1.4	143	3	0
Kernel3	1.582	3.4	848	3	0
Kernel4	8.101	7.1	2794	3	0
Kernel5	0.3	1.2	187	1	0
Kernel6	0.05	1.7	29	2	0
Kernel7	0.071	1.7	31	0	0
MapTest	44.974	26	4851	1	0
IteratorTest	0.34	1.1	208	0	0
MapDemo	0.07	1.7	26	0	0
JFE	290	70	9879	0	1
VCMPKiller	0.721	1.2	229	0	0

Table 6.2: The benchmark results of the analyzed programs for the context-sensitive, variables-names based, independent attribute variant of the algorithm.

Benchmark	**CPU Time (sec)**	**Space (Mb)**	**Struct**	**ERR**	**FA**
Kernel1	0.2	1.4	153	3	0
Kernel3	2.1	2.1	1745	3	0
Kernel4	4.196	3.7	3430	3	0
Kernel5	0.32	1.3	220	1	0
Kernel6	0.091	1.7	41	2	0
Kernel7	0.08	1.6	31	0	0
MapTest	56	14.1	12756	1	0
IteratorTest	0.37	1.1	208	0	0
MapDemo	0.08	1.6	26	0	0
JFE	1739	75.8	13833	0	1
VCMPKiller	0.561	1.1	440	0	0

Table 6.3: The benchmark results of the analyzed programs for the context-sensitive, variables-names based, relational variant of the algorithm.

Benchmark	**CPU Time (sec)**	**Space (Mb)**	**Struct**	**ERR**	**FA**
Kernel1	0.19	1.47	143	3	0
Kernel3	6.289	3.49	886	3	5
Kernel4	7.3	7.9	2794	3	0
Kernel5	0.3	1.29	187	1	0
Kernel6	0.05	1.74	29	2	0
Kernel7	0.07	1.72	31	0	0
MapTest	44.263	26	4851	1	0
IteratorTest	0.34	1.17	208	0	0
MapDemo	0.06	1.72	26	0	0
JFE	293	72.7	9878	0	1
VCMPKiller	0.43	1.19	208	0	0

Table 6.4: The benchmark results of the analyzed programs for the context-sensitive, allocation-site based, independent attribute variant of the algorithm.

Benchmark	**CPU Time (sec)**	**Space (Mb)**	**Struct**	**ERR**	**FA**
Kernel1	0.2	1.47	153	3	0
Kernel3	58.6	8.2	13410	3	5
Kernel4	3.345	3.75	3430	3	0
Kernel5	0.31	1.3	220	1	0
Kernel6	0.08	1.7	41	2	0
Kernel7	0.08	1.67	31	0	0
MapTest	55.39	14.3	12756	1	0
IteratorTest	0.38	1.17	208	0	0
MapDemo	0.08	1.67	26	0	0
JFE	1799	76.2	13832	0	1
VCMPKiller	0.5	1.2	415	0	0

Table 6.5: The benchmark results of the analyzed programs for the context-sensitive, allocation-site based, relational variant of the algorithm.

Benchmark	**CPU Time (sec)**	**Space (Mb)**	**Struct**	**ERR**	**FA**
Kernel1	0.18	1.47	143	3	0
Kernel3	8.8	4	2678	3	4
Kernel4	115.9	21.1	10402	3	14
Kernel5	1.453	1.3	275	1	0
Kernel6	0.05	1.7	29	2	0
Kernel7	0.07	1.7	31	0	0
MapTest	n/a	n/a	n/a	1	n/a
IteratorTest	0.4	1.19	198	0	0
MapDemo	0.06	1.7	26	0	0
JFE	n/a	n/a	n/a	0	n/a
VCMPKiller	1.42	1.2	274	0	0

Table 6.6: The benchmark results of the analyzed programs for the context-insensitive, variables-names based, independent attribute variant of the algorithm.

Benchmark	**CPU Time (sec)**	**Space (Mb)**	**Struct**	**ERR**	**FA**
Kernel1	0.21	1.48	153	3	0
Kernel3	1620	88	898035	3	4
Kernel4	6912	166.5	3040994	3	14
Kernel5	0.441	1.32	377	1	0
Kernel6	0.09	1.7	41	2	0
Kernel7	0.08	1.67	31	0	0
MapTest	n/a	n/a	n/a	1	n/a
IteratorTest	0.591	1.18	373	0	0
MapDemo	0.08	1.68	26	0	0
JFE	n/a	n/a	n/a	0	n/a
VCMPKiller	4.76	2.1	4097	0	0

Table 6.7: The benchmark results of the analyzed programs for the context-insensitive, variables-names based, relational variant of the algorithm.

Benchmark	**CPU Time (sec)**	**Space (Mb)**	**Struct**	**ERR**	**FA**
Kernel1	0.19	1.47	143	3	0
Kernel3	278	22.7	11110	3	14
Kernel4	n/a	n/a	n/a	3	n/a
Kernel5	0.37	1.3	212	1	0
Kernel6	0.05	1.7	29	2	0
Kernel7	0.07	1.7	31	0	0
MapTest	n/a	n/a	n/a	1	n/a
IteratorTest	1.24	1.19	325	0	0
MapDemo	0.06	1.7	26	0	0
JFE	n/a	n/a	n/a	0	n/a
VCMPKiller	0.5	1.2	227	0	0

Table 6.8: The benchmark results of the analyzed programs for the context-insensitive, allocation-site based, independent attribute variant of the algorithm.

Benchmark	**CPU Time (sec)**	**Space (Mb)**	**Struct**	**ERR**	**FA**
Kernel1	0.191	1.48	143	3	0
Kernel3	n/a	n/a	n/a	3	n/a
Kernel4	n/a	n/a	n/a	3	n/a
Kernel5	0.411	1.32	377	1	0
Kernel6	0.08	1.7	41	2	0
Kernel7	0.08	1.67	31	0	0
MapTest	n/a	n/a	n/a	1	n/a
IteratorTest	9.394	2.29	5985	0	0
MapDemo	0.08	1.68	26	0	0
JFE	n/a	n/a	n/a	0	n/a
VCMPKiller	1.51	1.48	1508	0	0

Table 6.9: The benchmark results of the analyzed programs for the context-insensitive, variables-names based, relational variant of the algorithm.

Chapter 7

Related Work and Conclusions

Several recent and ongoing research efforts [1, 41, 11] have goals similar, to varying degrees, to ours.

Predicate discovery Our approach is closely related to that of Ball et al. [1] and several similar research efforts [59, 9]. Like this prior work, we combine (i) identification of suitable abstractions of a program using powerful symbolic techniques (various combinations of weakest-precondition, theorem proving, and symbolic execution), with (ii) classical analysis techniques (abstract interpretation, model checking, and/or dataflow analysis) that utilize these abstractions. However, we put these ingredients together in a significantly different way.

First, 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 (*counterexample-driven abstraction refinement*). We do not claim that such "late" predicate discovery can be completely replaced by "early" predicate discovery. Rather, we believe that both may be employed together to improve efficiency, while retaining flexibility.

Second, our predicate discovery process attempts to generate predicates in such a fashion that an expensive relational analysis can be replaced by a cheaper independent attribute analysis without losing precision. (The theoretical foundation for this is the concept of *disjunctive completion* [13, 24].) This, for example, gives us the polynomial time algorithm for SCMP. In contrast, counterexample-based abstraction refinement relies on relational analysis, *even for distributive problems*, to generate counterexample paths. Relational analysis is, in general, exponential (in the number of predicates).

Third, Ball et al. work with propositional predicate abstraction, while we have shown how these ideas can be applied in the context of first-order predicate abstraction. Working with first-order predicate abstraction gives us more flexibility. For instance, the CMP algorithm we present relies on first-order predicate abstraction and can not be expressed in terms of propositional predicate abstraction (in a natural way).

Fourth, our approach combines predicate *discovery* and predicate *abstraction* (determining the update formulae for component methods) into one step, whereas in prior work, discovery and abstraction are done separately. Finally, we are not aware of any prior work showing how (staged) predicate discovery can lead to asymptotic improvements in the analysis efficiency without loss in precision.

Disjunctive Completion Our systematic derivation of the SCMP algorithm is similar to the process of *disjunctive completion*[13, 24]. 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 [24], 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., [35, 2, 6] for the weakest precondition computation and appropriate widenings [13] to ensure termination of the iterative process.

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 [11, 26, 1]; by requiring a finite bound on a portion of the infinite concrete state space to be explored [34, 33]; or by considering only temporal (event ordering) properties of the code [20].

In the Bandera project [11], 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* [10] such as SPIN [28] or SMV [43]. Projects similar to Bandera include Java PathFinder [26], JCAT [16], and Feaver [29].

Jackson and Vaziri [34] also use finite state-space exploration techniques to verify whether Java programs satisfy a specification written in a first-order language called Alloy [32]. 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 [33] 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 [41] and PVS [51]. ESC-Java relies on theorem-proving and program invariants to do verification. Traditionally, users had to provide the invariants, but recently there has been work on automatically "discovering" invariants [23] by generating candidate invariants heuristically, then eliminating those that are not irrefutable.

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.

Type Checking Other approaches to improved static checking includes languages with improved type systems [15]. We are not aware of any type system that can check for the kind of properties we have looked at in this paper.

Interprocedural Shape Analysis Noam Rinetzky [54] 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, 31, 18, 63, 19, 60]. 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.

Chapter 8

Further Work

In this chapter we report further work and suggestions for improving the front end and the CMP analysis in terms of performance and precision.

Front End Improvements

- A possibility to invoke Java methods from within a front end specification should be added. Method invocations will be used either for rule matching or for translation emition.
- The front end accepts the classes and the methods to analyze. Reachable method analysis can be performed to reduce the number of analyzed methods (by ignoring unreachable ones). As a result the number of generated predicates and the size of the generated TVP file will be reduced. This is especially true for partial program analysis.
- The front end provides a unique name to every local variable in the program. Every local variable translates to two or more predicates (using specification files for CMP). The front end can reduce the number of generated predicates by reusing some of them.
- Prior to emiting the translation, the front end backpatches labels that were not set by the translation generation phase. As a result, the whole translation is saved in memory, before being dumped to the file. Various optimizations can be applied to reduce the memory foot print of the front end.
- The front end generates nullification statements for local variables as they become dead. The same optimization may be performed for class fields.
- The front end performs the rule matching using static type information. The support for dynamic types need to be added.

- The front end does not support rules that manipulate variables of primitive types. Support for manipulating variables of primitive types need to be added.

- Slicing technique can be used to reduce the code analyzed. This can be done on various lavels, including the source and the TVP code.

Performance and Precision Improvements

- Support for multithreading and recursion neet to be added to the CMP analysis.

- When a variable or a field is assigned a value that is not maintained by the analysis, it is assigned an `unknown` value (to be conservative the analysis must assume that any possible value can be assigned). Current assignment of an `unknown` values is overly conservative. We suggest following improvements for dealing with `unknown` assignments:

  - Incorporating type information. The analysis may distinguish between variables of different types and treat the assignments of `unknown` values more precisely.
  - Trying to model variables of `java.lang.Object` type. Currently, any casting from a variable of `java.lang.Object` type is treated conservatively, since variables of `java.lang.Object` type are not modeled by the analysis (i.e., their values are not maintained by the analysis). This problem can be resolved by adding support for dynamic type information to the front end.
  - When an `unknown` value is assigned to a variable or a field, many predicates are being assigned an `unknown` value in the corresponding analysis. Techniques for lazy evaluation of `unknown` values may be applied to boost the performance of the analysis (i.e., by delaying the update of the predicates, the analysis, although working with the same precision, will be more scalable).

- Slicing can be incorporated into TVLA system to improve the performance of the analysis.

- Support for multithreading and inter-procedural analysis should be added to the TVLA system.

- It will be interesting to try the performance of the "vanilla" analysis in conjunction with an `invalid` instrumentation predicate.

- It may be interesting to try different TVP encodings for CMP. For example, one may use "vanilla" TVP with additional information such as general reachability (e.g., as proposed in [57]) or by adding more predicates to record valid iterators.

Bibliography

[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] 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".

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

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

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

[6] 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/.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

[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] 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.

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

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

[33] 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.

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

[35] 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.

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

[37] 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.

[38] 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.

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

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

[41] 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.

[42] 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.

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

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

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

[46] 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.

[47] 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.

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

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

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

[51] 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.

[52] 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.

[53] 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.

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

[55] M. Sagiv. http://www.math.tau.ac.il/∼msagiv, 2001.

[56] 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.

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

[58] 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.

[59] H. Saïdi. Model checking guided abstraction and analysis. volume 1824, pages 377–389. Springer-Verlag, 2000.

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

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

[62] 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.

[63] 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.

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

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

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

[67] 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.

Appendix A

Front End User Manual

As part of this thesis, We have developed a generic front end called JBT2TVP. The front end generates TVP files from a given Java program. The generation is based on specification files. One of the challenges in this front end is to allow analysis of partial programs which is used to obtain a feasible TVLA analysis of realistic programs (Other methods such as program slicing could also be applied). The challenge is to allow analysis of partial programs which is conservative under certain realistic assumptions.

The specification file describes the classes and the methods to analyze and supplies translation rules, which in turn, describe how to translate program statements. As a preprocessor step, the front end translates the Java program into an equivalent Jimple program[1]. Thus, the specification actually manipulates Jimple statements.

In the rest of this appendix we assume that the reader is familiar with TVLA and Jimple.

Each Jimple statement of the given program may be either ignored by the front end or translated to a local control flow graph; where nodes denote labels and edges are annotated with TVP actions. Local control flow graphs are later pasted together to form a TVP file.

In addition, JBT2TVP can also produce different sets that are used to control the set of generated predicates in the generated TVP files. These sets can also be used to parameterize the predicate-update formulae and other features of the generated analysis.

A.1 Functionality

The front end is invoked by the command:

```
jbt2tvp -rules <RuleFile1> [... <RuleFilen> ] -spec <SpecFile>
-outdir <output directory> [-nonull] -classes <Class1> [ ...
<Classn> ] [--app] [other Soot options]
```

[1] Jimple is a simplification of the Java bytecode [62]

where:

- The -rules option specifies names of rule files. At least one rule file has to be specified. Rule files describe the mapping between analyzed Jimple statements into TVP actions.
- The -spec option specifies a name for an xvm file containing information specific to the program being analyzed, such as analyzed classes and methods. In most examples, we use the ALL option to analyze all relevant classes.
- The -outdir option specifies output directory name.
- The -nonull option indicates that no optimization of assigning "dead" reference variables to null is performed. By default, this optimization is enabled since we found out that it drastically reduces the time and space of the generated analysis.
- The --app option enables the application mode where only one class file can be specified. All dependent classes will be automatically found and loaded. When this option is not enabled, all input classes have to be explicitly given as command line arguments and no automatic loading occurs. The actual bytecode is read from <Class$_1$>.class, <Class$_2$>.class, $\cdots$, and <Class$_n$>.class class files.

The following files are generated in the output directory:

- cfg.tvp contains a generated three valued program (TVP).
- sets.tvp contains set declarations usually used for predicate definitions and updates.
- cfg.log contains a log information related to a translation, e.g, the methods analyzed, warnings, unreachable methods, etc.

A.2 Running example

An example Java program is shown on Figure A.1.

Figure A.2 shows a corresponding Jimple program. An example of a rule file rules.xvm is shown in Figure A.5, in Figure A.6 and in Figure A.4 (This file was obtained from the general CMP specification by removing rules not needed to analyze this program). These rules are explained in Section A.4. An example of the file AppendToSet.xvm for this program is shown in Figure A.3. This file specifies that all classes and methods passed to the front end have to be analyzed and that the initial method of the analysis is the main method of the AppendToSet class.

The TVP output files after invoking the command

```
jbt2tvp -rules rules -spec AppendToSet -classes AppendToSet
```

```
import java.util.*;

class Benchmark {

    public static void main (String[] args) {
        Set s = new HashSet();
        s.add("value");
        Loop(s);
    }

    public static void Loop(Set set){
        Iterator i = set.iterator();
        while(i.hasNext()){
            String str = (String)i.next();
            set.add(str + "new");
        }
    }
}
```

Figure A.1: Example Java progam, serves as a running example

are shown in Figure A.7 and in Figure A.8. The log file is not shown here.

The file `cfg.tvp` containing the generated control flow graph is shown in Figure A.7. This figure begins with the generated initializations and then the generated TVP actions for `main` and `Loop` methods are shown.

In Section A.4, we explain the translation process.

The initialization part was generated as a result of applying rule 1 from the rulesfile.xvm. This rule specifies the TVP code that will be generated at the beginning of the `cfg.tvp` file. After the initialization, control is passed to the f1_l_0 label, which is the start label of the `main` method.

When a Jimple statement in Figure A.2 matches a rule in rules.xvm file, the translation associated with the matched Jimple statement is emitted. The matching is performed as follows:

- statement 1 matches rule 2. The variable `f1_$r1` matches `LHS` and the class `java.util.HashSet` matches `CLASSNAME`.
- statement 2 matches rule 6. The variable `f1_$r1` matches `OBJ`, the name of the function (`add`) matches `FUNCNAME` and the class `java.lang.Object` matches `PARAM1`.
- statement 3 matches rule 10. As a result the formal parameter is copied to a generated parameter using rule 9 and the "dead" local variable `f1_$r1` is nullified using rule 3.
- statement 4 matches rule 13.
- statement 5 matches rule 11 for copying a generated variable into the local variable and then uses rule 12 to nullify "dead" generated variable.
- statement 6 matches rule 7. The variable `f2_r1` matches `LHS`.

```
class Benchmark extends java.lang.Object
{
    public static void main(java.lang.String[])
    {
        java.lang.String[] r0;
        java.util.HashSet $r1;

        r0 := @parameter0: java.lang.String[];
        $r1 = new java.util.HashSet;
        specialinvoke $r1.<java.util.HashSet: void <init>()>();
        interfaceinvoke $r1.<java.util.Set: boolean add(java.lang.Object)>("value");
        staticinvoke <Benchmark: void Loop(java.util.Set)>($r1);
        return;
    }

    public static void Loop(java.util.Set)
    {
        java.util.Set r0;
        java.util.Iterator r1;
        java.lang.String r2;
        java.lang.StringBuffer $r3;
        java.lang.Object $r4;
        boolean $z1;

        r0 := @parameter0: java.util.Set;
        r1 = interfaceinvoke r0.<java.util.Set: java.util.Iterator iterator()>();
        goto label1;

     label0:
        $r4 = interfaceinvoke r1.<java.util.Iterator: java.lang.Object next()>();
        r2 = (java.lang.String) $r4;
        $r3 = new java.lang.StringBuffer;
        specialinvoke $r3.<java.lang.StringBuffer: void <init>()>();
        $r3 = virtualinvoke $r3.<java.lang.StringBuffer:
                     java.lang.StringBuffer append(java.lang.String)>(r2);
        $r3 = virtualinvoke $r3.<java.lang.StringBuffer:
                     java.lang.StringBuffer append(java.lang.String)>("new");
        r2 = virtualinvoke $r3.<java.lang.StringBuffer: java.lang.String toString()>();
        interfaceinvoke r0.<java.util.Set: boolean add(java.lang.Object)>(r2);

     label1:
        $z1 = interfaceinvoke r1.<java.util.Iterator: boolean hasNext()>();
        if $z1 != 0 goto label0;

        return;
    }
}
```

Figure A.2: The Jimple representation of the program from Figure A.1

```
ClientClasses = ALL
StartFunction = <Benchmark: main(java.lang.String[])>
FunctionsToAnalyze = ALL
```

Figure A.3: The specification file for the running example program shown in Figure A.1.

```
Sets = {
    UPDATENAMES = {add, addAll, remove, removeAll, ... }
    ModeledClasses = subtypeof(java.util.Collection) + subtypeof(java.util.Iterator)
}
TVLASets = {
    VarIterator = subtypeof(java.util.Iterator);
    VarCollection = subtypeof(java.util.Collection) + subtypeof(java.util.Map);
    Variables = ClientClasses;
   CallSites ={ };
   Labels  = { }
}
```

Figure A.4: Set declarations in the rule file for the running example program shown in Figure A.1.

- statement 7 matches rule 4.
- statement 8 matches rule 8.
- statement 9 matches rule 6.
- statement 10 matches rule 5.
- statement 11 matches rule 13 and uses rule 3 to nullify "dead" local variables.

Other Jimple statements of the input program were ignored by the front end.

A.3 The Specification File

The syntax of a AppendToSet program specification is shown on Figure A.9. The syntax is in extended BNF where A $\bowtie$ B denotes a (possibly empty) sequence of A's separated by B's.

Classes Classes modeled by the analysis. The **ALL** keyword stands for all classes passed in the command line. Whenever a class is specified as a modeled class, front end assumes that all ancestors of that class are also specified (it is required because a class can access fields of its ancestors). In the future the front end will derive this information automatically, or an option to specify the whole hierarchy of the class as modeled classes will be added.

```
// Rule 1
    [L1] Init [L9] =>
         L1 NewActivationRecord(global) $12
         $12 NewActivationRecord(curr) $13
         $13 Skip() start_label(StartFunction)

// Rule 2
    [L1] LHS = new CLASSNAME [L2] where
    (CLASSNAME.type subtypeof java.util.Collection || CLASSNAME.type subtypeof java.util.Map) =>
        L1 CreateNewCollection(LHS, curr, CLASSNAME) L2

// Rule 3
    [L1] LHS = null [L2] where
    (LHS.type subtypeof java.util.Collection || LHS.type subtypeof java.util.Map) =>
        L1 KillCollectionVariable(LHS, curr) L2

// Rule 4
    [Lb] goto [Lend] =>
        Lb Skip() Lend

// Rule 5
    [Lb] if VAR goto [Ltrue] [Lfalse] =>
        Lb Skip() Ltrue
        Lb Skip() Lfalse
```

Figure A.5: The assignment and control flow Jimple statements in the rule file.

StartFunction The first method analyzed by TVLA. The translation may refer to this method explicitly (e.g., initialize the analysis by transferring a control flow to the first label of this method).

FunctionsToAnalyze Member methods of the Classes, analyzed by the front end. The **ALL** keyword stands for all such methods.

A.4 Translation Rules

The syntax of translation rules' files are shown on Figure A.10 and Figure A.11.

`TVLASets` section allows to define sets, dumped into the `sets.tvp` file at the end of the translation. During the translation, additional elements can be added to the defined sets. The generated sets are usually used for predicate definitions and update formulae. If a special set called `Labels` is defined, the front end automatically fills it with labels used in the translation.

`Sets` section allows to define sets for usage inside the rule files. The ISIN operator is used to check membership in a set.

`StatementMap` section describes a mapping from Jimple statements into TVP actions.

Jimple statements in the rule files are called *modeled statements*. Every *modeled statement* has two parts: a syntax of a Jimple statement and a Boolean

```
// Rule 6
    [L1] OBJ.FUNCNAME(PARAM1) [L2] where
            FUNCNAME ISIN UPDATENAMES &&
            (OBJ.type subtypeof java.util.Collection ||
            OBJ.type subtypeof java.util.Map) =>
        L1 UpdateStructureDirectly(OBJ, curr, FUNCNAME) L2

// Rule 7
    [L1] LHS = OBJ.FUNCNAME() [L2] where
            FUNCNAME == iterator &&
            (OBJ.type subtypeof java.util.Collection ||
            OBJ.type subtypeof java.util.Map) =>
        L1 CreateIterator(LHS, curr, OBJ, curr) L2

// Rule 8
    [L1] LHS = OBJ.FUNCNAME() [L2] where
            FUNCNAME == next &&
            OBJ.type subtypeof java.util.Iterator =>
        L1 CheckIterator(OBJ, curr) L2

// Rule 9
    [L1] FGENPARAM = APARAM [L2] where
    (APARAM.type subtypeof java.util.Collection || APARAM.type subtypeof java.util.Map) =>
        L1 CopyCollectionVariable(FGENPARAM, global, APARAM, curr) L2
        PUT FGENPARAM IN VarCollection
        PUT APARAM IN VarCollection

// Rule 10
    [Lb] call FUNCNAME [Lsp] [Lep] [Lend] =>
        Lb Call([Lb]) Lsp
        Lep ReturnFromNonRecursive([Lb]) Lend
        Lep ReturnFromRecursiveFirst([Lb]) Lend
        Lep ReturnFromRecursiveNonFirst([Lb]) Lend
        PUT [Lb] IN CallSites

// Rule 11
    [L1] LHS = FGENPARAM [L2] where
    (LHS.type subtypeof java.util.Collection || LHS.type subtypeof java.util.Map) =>
        L1 CopyCollectionVariable(LHS, curr, FGENPARAM, global) L2
        PUT FGENPARAM IN VarCollection
        PUT LHS IN VarCollection

// Rule 12
    [L1] FGENPARAM = null [L2] where
    (FGENPARAM.type subtypeof java.util.Collection || FGENPARAM.type subtypeof java.util.Map) =>
        L1 KillCollectionVariable(FGENPARAM, global) L2

// Rule 13
    [Lb] return [Lend] =>
        Lb Skip() Lend
```

Figure A.6: The method invocation Jimple statements in the rule file.

```
// ---------- Initialization  ----------
    f0__l__0 NewActivationRecord(global) f0__l__0_2
    f0__l__0_2 NewActivationRecord(curr) f0__l__0_3
    f0__l__0_3 Skip() f1__l__0
// ----------  <Benchmark: void main(java.lang.String[])>  ----------
// Statement 1
    // f1_$r1 = new java.util.HashSet
    f1__l__0 CreateNewCollection(f1_$r1, curr, java.util.HashSet) f1__l__1
// Statement 2
    // interfaceinvoke f1_$r1.<java.util.Set: boolean add(java.lang.Object)>("value")
    f1__l__1 UpdateStructureDirectly(f1_$r1, curr, add) f1__l__2
// Statement 3
    // staticinvoke <Benchmark: void Loop(java.util.Set)>(f1_$r1)
    f1__l__2 CopyCollectionVariable(param_java.util.Set_0, global, f1_$r1, curr) f1__l__3
    f1__l__3 Call(f1__l__3) f2__l__6
    l_func_end_7104150 ReturnFromNonRecursive(f1__l__3) f1__l__4
    l_func_end_7104150 ReturnFromRecursiveFirst(f1__l__3) f1__l__4
    l_func_end_7104150 ReturnFromRecursiveNonFirst(f1__l__3) f1__l__4
    f1__l__4 KillCollectionVariable(f1_$r1, curr) f1__l__5
// Statement 4
    // return
    f1__l__5 Skip() l_func_end_4343712
// ----------  <Benchmark: void Loop(java.util.Set)>  ----------
// Statement 5
    // f2_r0 := @parameter0: java.util.Set
    f2__l__6 CopyCollectionVariable(f2_r0, curr, param_java.util.Set_0, global) f2__l__7
    f2__l__7 KillCollectionVariable(param_java.util.Set_0, global) f2__l__8
// Statement 6
    // f2_r1 = interfaceinvoke f2_r0.<java.util.Set: java.util.Iterator iterator()>()
    f2__l__8 CreateIterator(f2_r1, curr, f2_r0, curr) f2__l__9
// Statement 7
    // goto [?= f2_$z1 = interfaceinvoke f2_r1.<java.util.Iterator: boolean hasNext()>()]
    f2__l__9 Skip() f2__l__12
// Statement 8
    // f2_$r4 = interfaceinvoke f2_r1.<java.util.Iterator: java.lang.Object next()>()
    f2__l__10 CheckIterator(f2_r1, curr) f2__l__11
// Statement 9
    // interfaceinvoke f2_r0.<java.util.Set: boolean add(java.lang.Object)>(f2_r2)
    f2__l__11 UpdateStructureDirectly(f2_r0, curr, add) f2__l__12
// Statement 10
    // if f2_$z1 != 0[?= f2_$r4 = interfaceinvoke f2_r1.<java.util.Iterator:
                                                            java.lang.Object next()>()]
    f2__l__12 Skip() f2__l__10
    f2__l__12 Skip() f2__l__13
// Statement 11
    // return
    f2__l__13 KillCollectionVariable(f2_r0, curr) f2__l__14
    f2__l__14 KillIteratorVariable(f2_r1, curr) f2__l__15
    f2__l__15 Skip() l_func_end_7104150
```

Figure A.7: A TVP program generated from the Jimple program shown in Figure A.2. Only the modeled statements are shown.

```
%s VarIterator { f2_r1}

%s VarCollection { param_java.util.Set_0, f2_r0, f1_$r1}

%s Labels { l_func_end_4395840, l_func_end_4343712, f0__l__0, f2__l__9,
f2__l__8, f2__l__7, f2__l__6, f3__l__16_3, f1__l__5, f1__l__4, f2__l__15,
f1__l__3, f2__l__14, f1__l__2, f2__l__13, f1__l__1, f1__l__0, f2__l__12,
f2__l__11, f2__l__10, f3__l__19, f3__l__18, f3__l__17, f3__l__16,
l_func_end_7104150, f0__l__0_3, f0__l__0_2}

%s CallSites { f1__l__3}

%s Variables { f3_r0, param_this_Benchmark}
```

Figure A.8: Sets for a TVP program shown in Figure A.7

```
<clt> ::=     ClientClasses =           { <class>⋈, }
                                        | ALL [ − { <class>⋈, }]
              StartFunction =           <funcdecl>
              FunctionsToAnalyze =      { <funcdecl>⋈, }
                                        | ALL [ − { <funcdecl>⋈, }]
<funcdecl> ::= < <class> : id( <arg>⋈,) >
<arg> ::= java type
<class> ::= java class
```

Figure A.9: The syntax of a specification file.

condition after the **where** keyword. We also allow “abstract” Jimple statements such as assignments to unknown values which may be used to conservatively analyze partial Jimple programs.

We say that a rule *matches* an (abstract) Jimple statement if the syntax of the rule corresponds to the syntax of the statement and the semantic condition holds. In this case, the front end emits the associated translation after replacing the arguments with actual values. When more than one rule matches, the first rule in the first rule file is selected.

Each *modeled statement* has associated initial and final labels (some statements have more than two associated labels). Each initial label is a final label of the previous matched statement, and each final label is the initial label of the following matched statement. Additional labels, local to the current translation, may be used.

The associated translations use pattern variables such as **LHS**, which are substituted by the actual Jimple operands (variable names, class names, method names) during the translation. The translation can also use the type of the operand by using the **type** attribute. For example, `LHS.type` denotes the type of the left hand side argument. In the future more attributes may be supported.

The meaning of some specific abstract statements:

Init Abstract statement used as the initialization of the analysis.

```
<general_xvm> ::= [Sets = {<aliasdecl>* }]
                  [TVLASets = {<printdecl>* }]
                  StatementMap = {<opermap>* }
<aliasdecl> ::=   id= <set_decl>
<printdecl> ::=   id= <set_decl>
                  | id= <alias_name>
<alias_name> ::=  id
<set_decl> ::=    <class_set>
                  | { java type ⋈, }
<class_set> ::=   <class> | subtypeof(<class>)
                  | <class_set> + <class_set>
                  | <class_set> - <class_set>
<opermap> ::=     [id] <assignment> [id][where <boolexp> ]=> <trans>*
                  | [id] <goto> [id]=> <trans>*
                  | [id] if VAR goto [id][id]=> <trans>*
                  | [id] <switch> [id]=> <trans>*
                  | [id] call FUNCNAME [id][id][id]=> <trans>*
                  | [id] <return> [id]=> <trans>*
                  | [id] throw MESSAGE => <trans>*
                  | catch CLASSNAME on [id] with [id]=> <trans>*
                  | [id] Init [id]=> <init_trans>*
<boolexp> ::=     <primexp> | (<boolexp>)
                  | !<boolexp>
                  | <boolexp> && <boolexp>
                  | <boolexp> || <boolexp>
<primexp> ::=     <classexp> == <classexp>
                  | <classexp> != <classexp>
                  | <classexp> ISIN id
                  | <classexp> subtypeof <classexp>
<classexp> ::=    id | id.type | <class>
<init_trans> ::=  <trans>
<trans> ::=       <label> <action> <label>
                  | PUT <class> IN id
                  | PUT [<label> ] IN id
<action> ::=      id( <lab_class_id>⋈,)
<lab_class_id> ::= [<label> ]| <class> | id
<label> ::=       id | $id | start_label(<funcdecl>)
                  | start_label(StartFunction)
<funcdecl> ::=    < <class> : id( <arg>⋈,) >
<arg> ::= java type
<class> ::= java class
```

Figure A.10: The syntax of the translation rules. The <assignment> rules are shown in Figure A.11.

<assignment> ::= LHS = RHS
| LHS = RHSOBJ.RHS
| LHS = RHSCLASS.RHS
| LHSOBJ.LHS = RHS
| LHSCLASS.LHS = RHS
| LHS = new CLASSNAME
| LHS = unknown
| LHSOBJ.LHS = unknown
| LHSCLASS.LHS = unknown
| LHS = null
| LHSOBJ.LHS = null
| LHSCLASS.LHS = null
| FGENPARAM = APARAM
| LHS = FGENPARAM
| RGENPARAM = RETURNV
| LHS = RGENPARAM
| FGENPARAM = null
| RGENPARAM = null
| [LHS =] OBJ.id($\text{PARAM}_i \bowtie$,)
| LHS = CAUGHTOBJECT

Figure A.11: The <assignment> syntax rules.

LHS = RHS Left hand side local variable is assigned a value of a right hand side local variable. This statement may be also used to model the behavior of the cast statement, using the `where` clause and specifying appropriate types for the LHS and RHS pattern variables.

LHS = null This statement is used for two purposes: (i) it is evaluated whenever a Jimple statement assigning a null value to a variable is encountered. (ii) whenever the kill dead variables optimization is enabled, this statement is used to nullify dead variables. For generated variable names (passed as actual parameters or returned from a method) `FGENPARAM = null` and `RGENPARAM = null` statements are used instead.

LHS = unknown A local variable is assigned an unknown value. This is an abstract statement used for conservative analysis of partial programs (e.g., when a modeled variable receives a value from an unmodeled method invocation, it receives a conservative value).

switch The modeling of the switch statement is currently conservative. There is no way to check the value of the switch condition, but only to transfer the control flow from the begining of the switch statement to the begining of each case statement.

call FUNCNAME Whenever a call to a an analyzed method is encountered the front end translates this call as follows:

1. The actual parameters are copied to a newly generated variables via the `FGENPARAM = APARAM` statement. Here, `FGENPARAM` denotes name

of a new generated formal variable, `APARAM` is the actual parameter.

2. Virtual functions are handled by generating a translation associated with the `call FUNCNAME]` abstract Jimple statement for all possible functions. Here the front end uses Soot's class hierarchy analysis to locate the possible invoked functions.

3. The return value is assigned by the translation associated with the abstract statement `LHS = RGENPARAM`.

The prolog of every procedure can copy the formal parameters into local variables using the translation associated with `LHS = FGENPARAM` abstract statement.

return Whenever a return Jimple statement is encountered front end translates the return from a method in the following way:

1. The return value is assigned to a newly generated temporary by the translation associated with the abstract statement `RGENPARAM = RETURNV`.

2. The control flow is transfered to the last label of the procedure using the translation for the [return] statement.

OBJ.FUNCNAME() This Jimple statement matches a call to a function whose return value is ignored. This statement is again useful to analyze partial code when the code of the invoked mathod is unavailable.

LHS = OBJ.FUNCNAME() This Jimple statement matches a call to a function whose return value is assigned to a variable.

The front end issues a warning for every class in the analyzed classes that contains at least one static field of a modeled class. The modeled classes are classes inside the analyzed classes or the ModeledClasses set (if defined). The warning may indicate that additional statements need to be added to the Init translation (e.g., creating an object for every specified class).

There is no way to define translations for local variable declarations or field declarations. The TVLASets may be filled with variables' and fields' names by initializing the sets with appropriate types of variables and fields. Variables' names may be later added to TVLASets by using the `PUT id IN id` operator inside a translation rule.

The front end does not currently allow to extract all the Jimple information. We are aware of the following Jimple information which is currently unavailable:

- Dynamic type information of variables and fields cannot be extracted.
- Return type of method calls cannot be extracted.
- No support for primitive type variables.
- No built-in support for multithreading. Multithreading can be explicitely modeled in the rule file.

- Switch statement is treated conservatively.