Practical Runtime Universe Type Inference

Marco Bär

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

November 16, 2005 — May 16, 2006

Supervised by:
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The Universe type system provides means to structure the heap memory into so-called Universes. This structuring makes it easier to reason about object structures on the heap by assigning all references an additional Universe type. These additional types specify the access rights of the holder of such a reference. In this work, we extended an existing Runtime Inference tool that infers Universe annotations for existing Java programs through runtime observation of a test case.

We have added the possibility to handle arrays and static methods. Furthermore, we developed a method body inference algorithm using abstract interpretation of the types on the method's operand stack and registers. The abstract interpretation is a static analysis that uses the results of the runtime inference as input. The tool is now able to infer types of local variables, new-statements, static method invocations, and it is able to insert Universe type casts where it is necessary.

To improve the code coverage of our Runtime Inference Tool, we have added the possibility to combine several test cases. The dynamic heap structure of the different test cases is globally harmonized and Universe annotations for the source code is inferred. The possibility to join several test cases increases the quality of the inferred annotations significantly.

Acknowledgments

First of all, I would like to thank my girlfriend Serena for supporting me during the course of this thesis. Especially, I would like to say ”sorry” for all the time I have spent working on this thesis instead of spending time with you.

Special thanks to my supervisor Werner Dietl for taking so much time to help me with this thesis, for his useful advice and comments on the report.

Thanks to Michael Szönyi for the peer review of this report.

Finally, I would like to thank my parents for supporting me throughout my studies at the ETH and for making my life beyond school as easy as possible.

Contents

Definition of Terms

Term	Explenation
EOG	Extended Object Graph.
JVM	Java Virtual Machine.
JVMTI	Java Virtual Machine Tooling Interface.
JNI	Java Native Interface.
JDK	Java Development Kit.
JML	Java Modeling Language.
Array Component	A component of an array. An array object contains of a number of variables called *components*.
Field	*Fields* are variables of a class type (either instance or class variables).
Instance Variable	A field that is not declared static is called an *instance variable*.
Class Variable	A field that is declared static is called a *class variable*.

Table 1: Notations. Most of the definitions are taken from the Java Language Specification [13] or the Java VM Specification [25].

Chapter 1

Introduction

The Universe type system provides means to structure the heap memory into so-called Universes. This structuring makes it easier to reason about object structures on the heap. The structuring of the heap is performed by assigning each reference in the program an additional *Universe type* (also referred to as *Ownership modifier*). The Universe type system will be described in more detail in Section 1.1.

Runtime Universe type inference is trying to infer these Ownership modifiers for all variables of a given Java program by examining the execution of a test case of this program. The representation of the dynamic heap structure that was generated throughout this test case has to be mapped to the static structure of the source code. This way, we may gain insight on how well the Universe type system can be applied to real-world problems, i.e. how well it maps to the dynamic object structure on the heap. On the down-side however, runtime inference has to deal with code coverage issues like any other test based approach. Bad code coverage may lead to incomplete or even wrong inference of Universe types.

The goal of this Master thesis is to bring the existing Runtime Inference tool developed by Frank Lyner [19] to a stage where no restrictions on the input program are necessary and the Inference tool produces the desired output. The Inference tool should produce output that can be used to annotate existing Java source code. The annotated source code should then be compilable by the MultiJava[7] compiler into which the Universe type system was integrated.

1.1 Universe Type System

Aliasing

In object-oriented programming languages, every access to an object is done through a reference. An inevitable effect of this mechanism is the introduction of aliases. We speak of *aliasing* when two or more variables hold a reference to the same object. In some cases this is voluntary and brings benefits (e.g. parameter object can be passed by reference and do not need to be copied), in other cases aliasing is not wanted and introduces difficulties. This is why every object-oriented programming language has a notion of *access modifiers* which regulate the access of variables to some extent. However, the common access modifiers for Java (public, protected, package, private) have proven not to be powerful enough to really take advantage of formal methods such as invariants or pre- and postconditions, not speaking of even more sophisticated ones. Two common examples where Java access modifiers fail due to aliasing are *reference leaking* (Listing 1.1) and *reference capturing* (Listing 1.2).

Reference leaking is the leaking of implementation details to the outside. In the example, a reference to the private array `b` is passed to the caller of the method `getB()`. A hostile caller can now simply break `A`'s invariant by assigning a value greater than 10 to an array component. In order to prove the invariant, it is therefore not enough to show that the class invariant of `A` is preserved by all methods of `A` itself. Additionally, it needs to be proved that all methods of all classes in

the system preserve the invariant, which is definitely not desirable. Reference capturing has the same effect as leaking. The mechanism of getting the reference to the internal representation of the structure is different and can be seen in the second example.

Listing 1.1: Reference Leaking. The public method getB() leaks a reference to the private array b. A caller of getB() can break A's invariant.

```
class A{
        //invariant: b[i] <= 10, for all i
        private int [] b;          //private object should not be modifiable from the outside

        public int [] getB(){
                return b;                  //leaks b
        }
}

class C{
        public void violate1(A a){
                int [] b= a.getB();
                b[1]= 12;          //breaks A's invariant
        }
}
```

Listing 1.2: Reference Capturing. Method setB() captures the passed reference. The caller of the method still has access to the private array b and can break A's invariant.

```
class A{
        //invariant: b[i] <= 10, for all i
        private int [] b;

        public void setB(int [] newB){
                for (i=0; i< newB.length; i++){
                        if (newB[i] > 10){       //Checks the invariant
                                return;
                        }
                }
                b= newB;                   //capture newB
        }
}

class C{
        public void violate2(A a){
                int [] b= new int[2];
                b[0]=1;
                b[1]=2;
                a.setB(b);
                b[1]= 12;          //breaks A's invariant
        }
}
```

Universes

The central idea of the Universe type system is the structuring of the heap into Universes (generally referred to as *contexts*). Every object belongs to a context and each context has at most one

owner. The whole hierarchy is rooted at the *root context* which is the only context not owned by a particular object. All objects that are created from the main() method are added to this context and have no owner, all other objects belong to the owner of the context they belong to. This idea originates in the work *The Geneva convention on the treatment of object aliasing* [15] by Hogg et al.

In the Universe type system, each reference needs to be assigned a Universe type that is dependent on the relationship between the source and the target of the reference. There are three Universe types describing this relationship:

- **peer** The source of the reference and the target of the reference are in the same context.
- **rep** The target of the reference is in the context owned by the source.
- **readonly** Nothing can be said about the relationship of the two objects.

Write access is only granted for **rep** and **peer** references, but not for **readonly** ones. I.e. write accesses across context boundaries are not allowed.

Subtyping

The subtype relation on Universe types follows the subtype relation in Java. Every **rep** and **peer** type of a class is a subtype of the **readonly** type of the same class. If one class is a subtype of another class in Java, it is also a subtype in the Universe type system, if its Universe type is in a subtype relationship to the Universe type of the other class (see Figure 1.1).

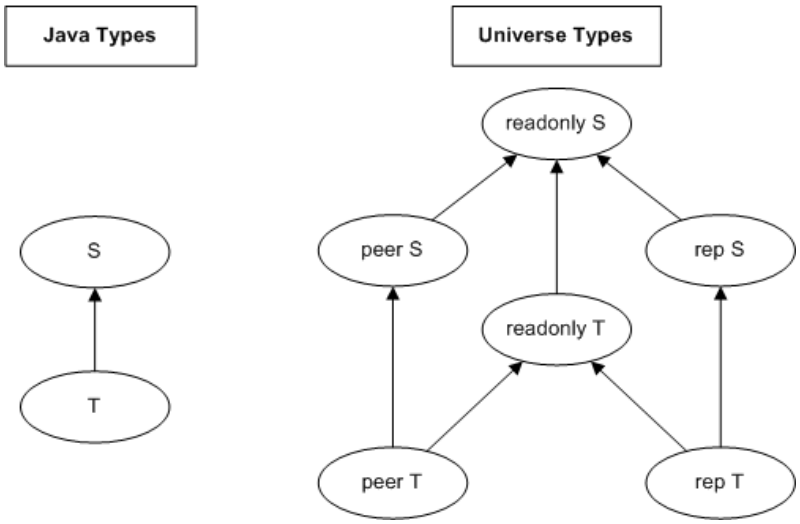

Figure 1.1: Subtyping relationship between Universe types. T is a subtype of S; **peer** and **rep** are subtypes of **readonly**.

Type Combinator

In order to determine the resulting Universe type of a dereferencing chain, e.g. a field access on an object such as x.f, the type modifier of the first object and the type modifier of the second object have to be considered. For this, the type combinator matrix in Table 1.1 is applied. The Universe type of the first object is displayed in the first column, the Universe type of the second object in the top row. For the previously mentioned field access x.f, x is the first object and f the second. The type combinator is also used to get the Universe type of method arguments or return types, because they have to be interpreted relative to the target object.

*	peer	rep	readonly
peer	peer	readonly	readonly
rep	rep	readonly	readonly
readonly	readonly	readonly	readonly

Table 1.1: Type Combinator of the Universe Types. The **this** reference is always a **peer** reference. If the first parameter is a **this** reference, the type combinator is not applied.

Method calls

In general, a method call on an object is considered to be a write access, because the method may change the state of the object. However, this would mean that no methods can be called on **readonly** objects, which is far too restrictive. This is why the notion of **pure** methods was introduced. **Pure** methods may not change the state of any existing object, they may only call other **pure** methods, and all parameters must be **readonly**.

Instance methods always run in the context of the target object. Since static methods do not have a target object, their execution context needs to be specified upon the invocation of the method. Static methods may be executed either in the **peer** or **rep** context of the caller method. A more detailed description of static method calls follows in Chapter 3.

Incorporation into Java

The Universe type system is integrated into the MultiJava compiler [7] and the Java Modeling Language (JML) Tools. Existing Java source code needs to be annotated with Universe types and can then be compiled with the MultiJava compiler. The generated bytecode can be run by a normal Java Virtual Machine; all Universe type checks are added using normal bytecode that is injected (e.g. a non-**readonly** check on a target before method invocation).

1.2 Goal

Lyner [19] developed the basic algorithms and provided a prototypical implementation of the Runtime Inference algorithm. The goal of this thesis is to extend his work to be practically usable. This means that there should be no restrictions on the input program of the tool. Namely, array annotation, static method handling, and method body annotation have to be implemented. The code coverage of test cases should be improved, such that our tool produces good results.

1.3 Outline

In Chapter 2 we describe the Inference tool developed by Lyner during his Master thesis [19] and point out its limitations. In Chapter 3 we present a way to handle static methods. With the chosen approach, we can successfully determine the execution context of a static method and the Universe type of newly created objects. In Chapter 4 we present array annotation using the bytecode instrumentation features of the Java Virtual Machine Tool Interface (JVMTI). Furthermore, in Chapter 5 we will present an approach for method body inference that is based on abstract interpretation. The types on the operand stack and in the method frame registers are interpreted abstractly and the Universe annotation of method body statements (e.g. local variable declarations, new-statements, etc.) are inferred. A method to increase the code coverage of our tool by joining multiple test runs is presented in Chapter 6. Chapter 7 concludes this thesis with the discussion of a set of test runs with the Inference tool, the presentation of related work, and the suggestion of future work on the Runtime Inference program.

Chapter 2

Runtime Inference Tool

In this chapter we will explain the basic algorithm of the Runtime Inference tool implemented by Frank Lyner. The algorithm overview is presented in Section 2.1. Some implementation details that help understanding the rest of this report are outlined in Section 2.2. We will also point out the limitations of his implementation to which we will refer to as *Version 1* in Section 2.3.

2.1 Algorithm Overview

The basic algorithm consists of the following steps:

1. Information gathering
   (a) Monitoring program execution and building a log file (Tracing agent).
   (b) Building data structure (Extended Object Graph) from log file.
2. Structuring the object store in Universes.
3. Finding valid annotations.
4. Generating output.

The information is gathered using a *Tracing agent* that is built on top of the Java Virtual Machine Tool Interface (JVMTI). It is a native interface part of the Java Development Kit (JDK) that allows to inspect the state and control the execution of applications running inside the Java Virtual Machine. Since the interface of the JVMTI is native, the code of the Tracing agent is written in C++. The Tracing agent creates a log of a program execution and stores it in an XML file format. The log is event based and consists of events such as method entry, method exit, field update and so on.

The produced log is parsed and processed by the so-called *Type inferer* which performs all other steps of the algorithm. The Type inferer (written in Java) builds up an Extended Object Graph (EOG) from the events in the log. The EOG is an overlay of all object graphs of a program execution (see Figure 2.1). A simple object graph is a snapshot of the heap at a certain point in time. The nodes of the object graph represent objects on the heap; vertices represent either *variable references* or *write references*. Variable references are used to denote that an object is stored in a variable of another object. E.g. if object `b` is stored in a field of object `a`, a variable reference from `a` to `b` is added to the graph. Each event in the log will trigger some manipulations of the EOG. For example, a non-pure method call from object `c` to object `a` is considered to be a write access from `c` to `a`. Therefore, a write reference from `c` to `a` is added to the graph.

Once the EOG is completely built up, the Type inferer structures the object store into Universes. Every object must be assigned to exactly one owner. Lyner used the well researched notion of *dominators* for flowgraphs as a first approximation for the owner of an object, this property

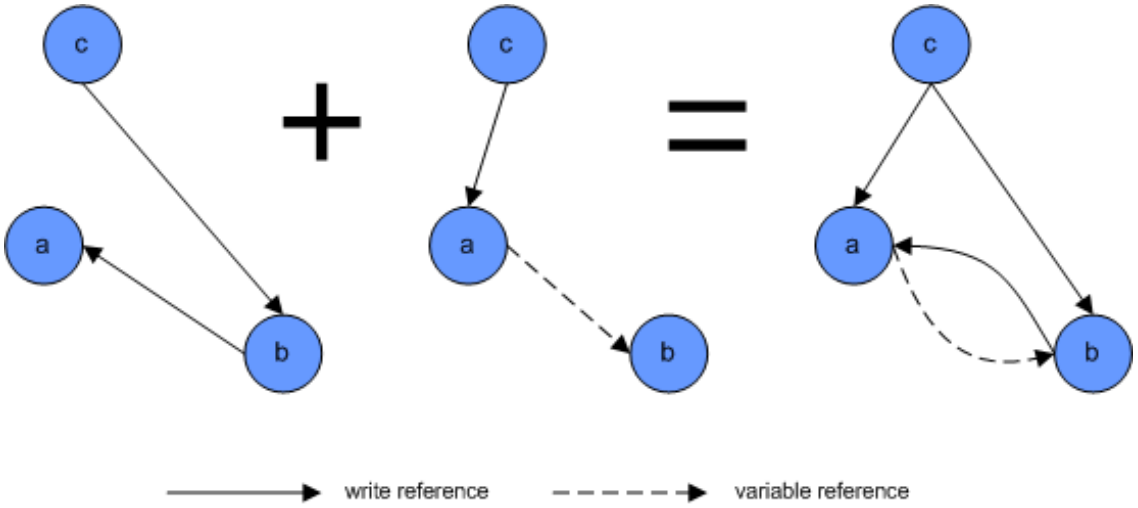

Figure 2.1: Extended Object Graph. Overlay of two object graphs with write references denoting write accesses from one object to another and variable references denoting that an object is stored in another object's variable.

is known as *owners as dominators* [6, 4, 5]. In the Universe type system, the root context is not owned by any object. However, to make sure that the data structure stays consistent and every object has an owner, an artificial object that is the owner of the root context is introduced. The property of the Universe type system that no write reference may cross any context boundary (write accesses are only allowed in **rep** or **peer** relationships) is enforced and possible conflicts are resolved until a valid hierarchy is established.

Once all conflicts in the graph are resolved, the annotation step is performed. For each variable (e.g. fields of classes, method return values, and method arguments), every variable reference in the graph is followed. The contexts to which the target and the source of these references belong to are inspected. If both ends belong to the same Universe, a **peer** annotation can be inferred, if the starting object of the reference is the owner of the end object, a **rep** annotation can be inferred. If the reference connects two objects in arbitrary Universes, nothing can be said about their relationship and a **readonly** annotation must be inferred.

In the end, the found annotations are stored in an XML file format that conforms to the annotations.xsd[1] schema. This file can be used by Marco Meyer's annotation tool [20] to annotate existing Java source code.

2.2 Implementation

Since we needed to make some additions to the implementation during the course of this thesis, we present some implementation details of Version 1. As mentioned before, the implementation is divided into two parts. The first part (called *Tracing agent*) is built on top of the JVMTI and is used to produce the trace file. The second part (called *Type inferer*) is the actual type inferring program that builds up the EOG, infers the Universe types, and writes the found annotations to a file.

The Tracing agent makes use of the callback functions provided by the JVMTI. The agent can specify functions that will be called if a certain operation was performed by the Java Virtual Machine. These functions are used to extract the necessary information from the JVM and generate xml tags for the output file. We will refer to one of these tags as *event*. The events stored in the Tracing agent output file are then parsed by the Type inferer and the EOG is built up. The

[1] see Appendix C

following is a list of events that are used by Version 1 and it is stated what modifications on the EOG are needed:

Method Entry If the method that is entered is non-pure, the event is considered to be a write access on the target of the call. A write reference from the caller to the target is added to the EOG. Additionally, all parameters become accessible to the target object. For each parameter, a variable reference is added, starting at the target object and ending at the parameter object.

Method Exit If the method does not have a return value of reference type, the EOG is not modified. Otherwise, a variable reference is added, starting from the target object and ending at the returned object.

Class Prepared A class is prepared for loading. This will not have any impact on the graph. All methods and fields of this class are parsed and made ready for further use by the algorithm.

Field Modification A field modification is treated as a write access on the target object. A write reference from the caller to the target is added. A variable reference from the target object to the object assigned to the field is added as well.

The implementation of the Type inferer (the Java part) is built around the representation of the EOG (see Figure 2.2). The class Graph is used to store all information about the EOG. That is, objects (class GObject) and references, which can be either variable references (class VarReference) or write references (class WReference). The same class is also used to store the variables to be annotated, such as parameter variables and field variables.

The different steps of the algorithm are implemented as visitors of the Graph class (see Figure 2.3). The visitor approach has the benefit that additional steps can be easily added by simply plugging in an additional visitor. Furthermore, it is not hard-coded which visitors are applied to the graph, it can be specified by a configuration file. This makes it possible to run the algorithm with optional visitors as well. The configuration file is also used to specify the input files for the algorithm (e.g. the trace file, the file specifying the pure methods in the system, and the output filename). Version 1 used the following visitors (in the order they are applied to the graph):

BuildUpVisitor builds up the graph by reading in events from the log file and performing the necessary modifications to the EOG.

DominatorVisitor applies the dominator algorithm by Lengauer/Tarjan [17] to the EOG. This computes the direct dominator for each object and the owner of each object is set.

StoreDominatorLevelVisitor computes the depth of each object in the dominator tree.

ResolveConflictsVisitor resolves possible conflicts that were introduced by the ”owner as dominator” approximation. Conflicts originating by write references crossing context boundaries are also resolved.

HarmonizationVisitor maps the dynamic structure of the EOG to the static structure of the program classes. Inconsistent references are harmonized and the annotations are found for each variable. Harmonization may be necessary, for example, if not all of the objects that are stored in a variable belong to the same context.

OutputVisitor writes the found annotation to an XML file using the Apache XMLbeans package [23].

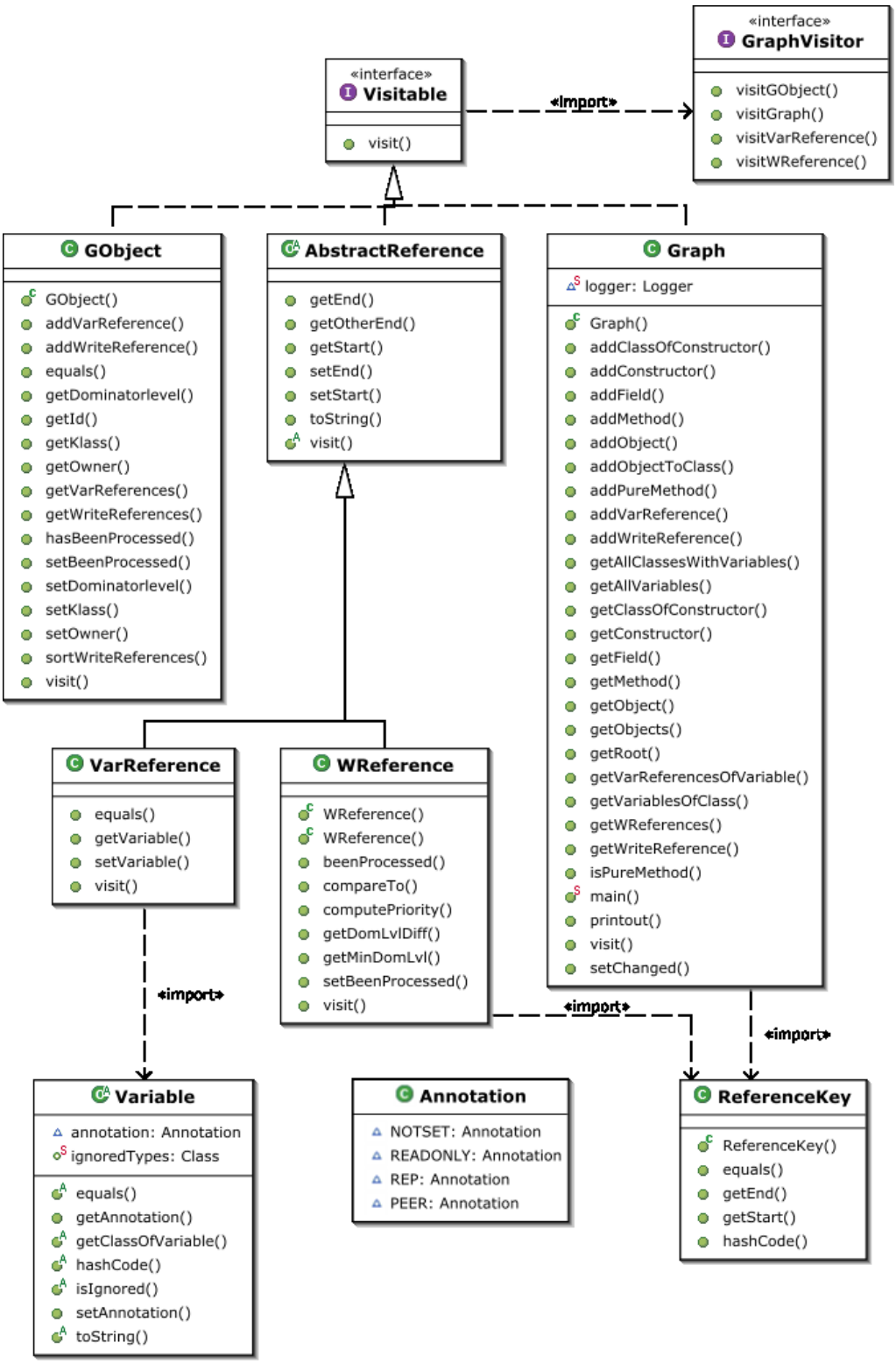

Figure 2.2: UML diagram of the graph package with the classes used to build up and maintain the EOG. GObject is the representation of a Java object. There are two kinds of references: VarReferences denote a variable (field, parameter, return type) relationship, WReferences denote a write access from one object to another.

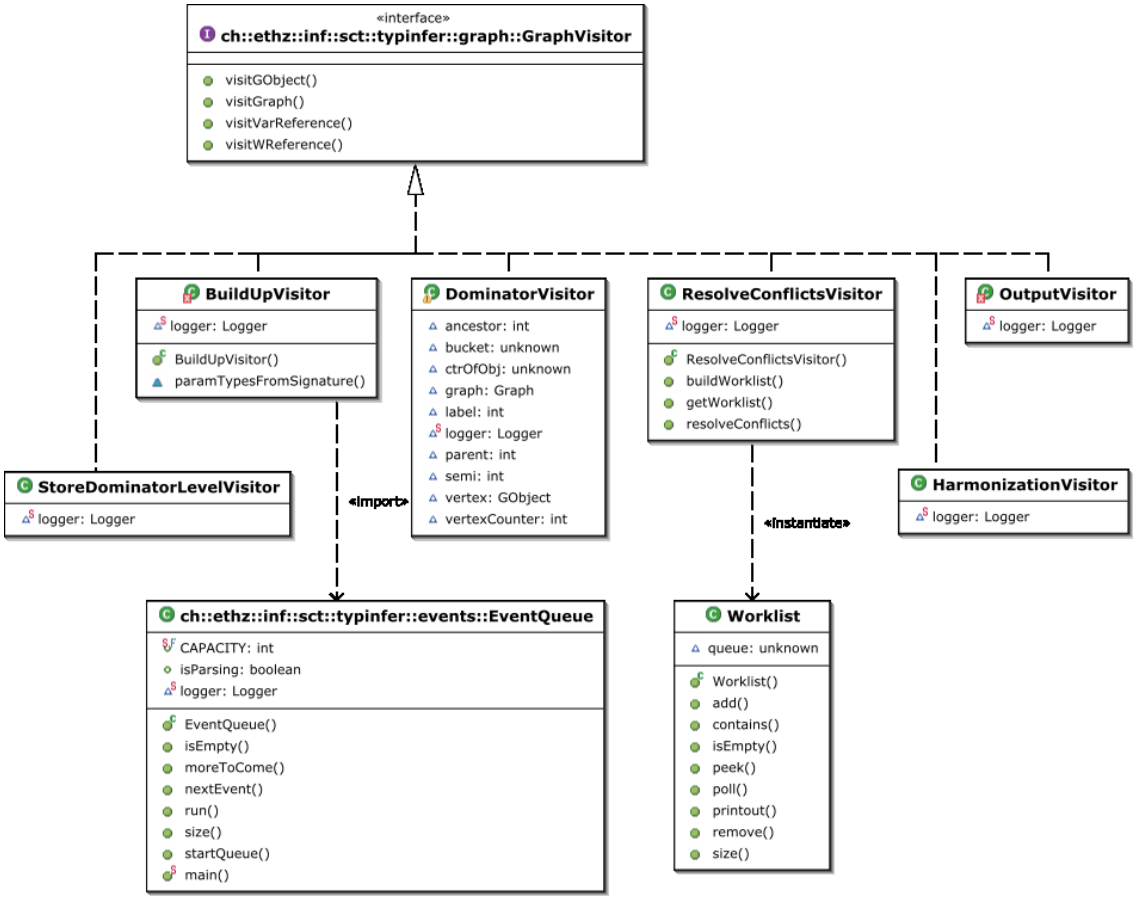

Figure 2.3: UML diagram of the algorithm package which includes all visitors. Each visitor stands for a particular step in the algorithm and visits the Graph object. The EventQueue is used to parse the tracing information stored in the input xml file generated by the Tracing agent.

2.3 Limitations of Version 1

Version 1 imposed some restrictions about the supported language structures. It is the goal of this thesis to loosen up these restrictions such that the Runtime Inference tool can be used for any Java program.

Static methods were handled in a simplified way because the lack of a target object did not allow to let them be handled like instance methods. We will show how they can be dealt with in Chapter 3. Arrays were not addressed because of some limitations of the JVMTI, we will present a solution to this problem in Chapter 4. Furthermore, the scope of the project was restricted to programs with only one thread. We did not make any more investigations about the implications of multiple threads in a program, but our implementation of static methods might lead to some problems there, which are mentioned in Section 3.3.

The annotation of method bodies, i.e. local variables, object creations, and casts, was also beyond the scope of the original project and will be discussed in Chapter 5. For this problem, we combined the approaches of static and runtime inferences by using an abstract interpretation of the types on the method argument stack which uses the static bytecode information.

Chapter 3

Static Methods & Object Creation

In Version 1 of the Inference tool, static method calls were treated to be running inside the root context. Conceptually, this is only correct for the static `main()` method of each program, because any other static method will not necessarily run in the root context. However, this simplification is valid if all static methods neither create any objects nor modify any of their parameters. Of course these assumptions cannot be made for regular programs, so we had to find a new approach to solve this problem. The reason why static methods need to be handled differently is that there is no target object of the call. As we implemented static method handling, we discovered a simple way to annotate object creations (i.e. new-statements). In Section 3.1 we take a look at how instance methods are handled. The way we handle static methods that requires only minor changes of Version 1 is presented in Section 3.2. In Section 3.3 we take a look at our implementation. Finally, in Section 3.4 we show in what form the output needs to be generated.

3.1 Instance Method Calls

An instance method call of a non-pure method on object `b` from object `a` is treated like a write operation from `a` to `b` (see Figure 3.1). This is indicated by a write reference from `a` to `b` in the Extended Object Graph. Pure methods are not allowed to modify any existing object. Therefore, a pure method call is not treated as a write operation and no write reference needs to be added. In any case, all parameters of reference type will become accessible from the target object `b`, hence variable references from `b` to the parameters (`p1` and `p2`) are added. Return values of reference type are handled like parameters. Of course, write operations within the called method will yield write references from object `b` to the altered (or newly created) objects such as `o`. This means that the called method will execute relative to the target object.

The Universe modifiers of the parameter and return types are deduced from the relationship between the target object of the call and the parameter/return objects. In the example 3.1, a **peer** annotation for `p1` and `p2` would be inferred, because they belong to the same context as `b`.

3.2 Static Method Calls

For static method calls, there is no target object because static methods do not belong to a specific instance of a class. This implies that **rep** types cannot be used in their signature, for local variables, or anywhere else within the method. **Rep** can only be interpreted relative to an object currently in charge, which is not present for static methods. Unlike for instance methods, which are executed relative to the target object, the execution context of static methods needs to be specified by the programmer. Either the static method is executed in the **peer** context of the caller or in the context owned by the caller (**rep**). Argument and return types of the called static method are then interpreted relative to the context in which the called method executes. Listing 3.1 shows a class

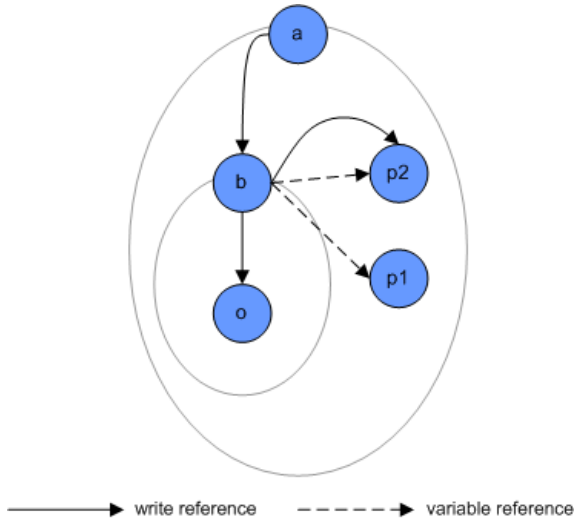

Figure 3.1: Instance method call from `a` to `b` with parameters `p1` and `p2`. Object `o` is created by the called method and will therefore be added to the context of `b`. For `p1` and `p2`, the algorithm would infer a **peer** annotation, even though there is no write reference to `p1`.

X with a static method returning a new object. Calls to this method can be annotated in two ways (this annotation is called the *invocation type* of the call).

- Without Ownership modifier, this means the method is called relative to the current context (line 11). The returned object becomes **peer** to the caller.
- With the modifier **rep**, this means the method is called relative to the context owned by **this** (line 15). The returned object becomes **rep** to the caller.

A **readonly** annotation of a static call (either pure or non-pure) is not allowed. This would not make sense because if there are any objects created within that static method, it will be undefined which context this object needs to be added to. Objects can be created within pure or non-pure methods.

Listing 3.1: Two possibilities of static method invocation.

```
1   class X{
2       static peer Object newObject(){
3           return new peer Object();
4       }
5   }
6
7   //from somewhere else:
8
9   //peer call
10  peer Object peer_obj;
11  peer_obj= X.newObject();
12
13  //rep call
14  rep Object rep_obj;
15  rep_obj= rep X.newObject();
```

As shown previously, all parameter and return objects of reference type of an instance method call are connected to the target object by a variable reference. Again, there is no such object for

static calls that could be the source of such a reference. In Version 1 of the Inference tool, the artificial root object that is the dominator of all objects in the EOG is used as the target object of any static call, which is, as we pointed out earlier, only correct for the static main() method. Objects created by a static method should be added to the context specified by the invocation type (as shown in Listing 3.1). A better way to treat static method calls that is correct in all cases has to be found.

Artificial Target

To overcome the lack of a target object, an artificial object which is described in more detail in Section 3.3 is inserted into the EOG for each static method call. These artificial objects can now act as target objects and static calls can be handled almost like any instance method call. The algorithm needs to make sure that no type in the signature of a static method is **rep** and that the call itself is annotated either with **peer** or **rep**. The same is true for **pure** static methods, the only additional constraint for these methods is that parameter types may only be **readonly**, just like for pure instance methods. We can now review the handling of static *method entry*, static *method exit* and *write operation* events within static methods to see how the EOG needs to be modified upon these events, such that we can ensure the properties mentioned above.

Method Entry To distinguish between **rep** and **peer** calls, a variable reference representing the method call needs to be added to the EOG. This variable reference connects the caller object a with the artificial target object X (Figure 3.2) and ensures that the call will be annotated in the harmonization and annotation phase just like any other variable. To ensure a non-**readonly** annotation of the call, a write reference to the artificial object is introduced. This will cause the object to be either in the **peer** or **rep** context of the caller, since write references can never cross context boundaries. This write reference is also needed for **pure** static methods, because a static method call can never be annotated with **readonly**. The **readonly** annotation of **pure** method parameters is not affected by this write reference as it does not connect any parameter object with the artificial target object.

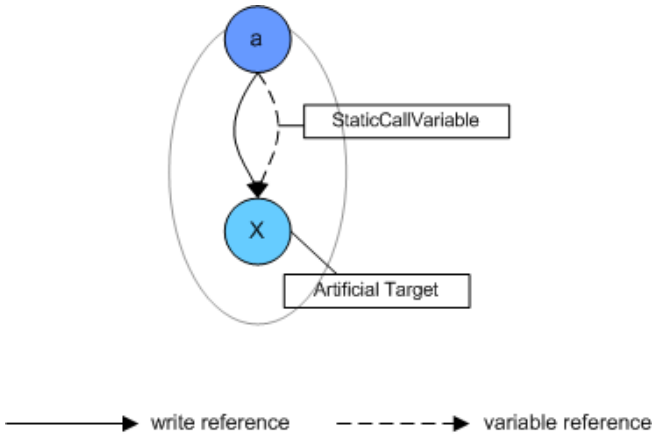

Figure 3.2: Static method call. Caller a is connected to the artificial target X, representing the static method call, by a write and a variable reference (StaticCallVariable). The variable reference is used to infer the invocation type of the call and the write reference ensures a non-**readonly** annotation.

Write operation A write operation inside a static method (i.e. field modification or non-pure method call) can be handled almost like other write operations. The artificial object is used as the source of the write reference that needs to be added to the EOG. This is correct, because

the artificial object will be in the same context where the static call was specified to execute (either **peer** or **rep** to the caller of the method). Since **rep** annotations are not allowed inside static methods, a **peer** annotation of the altered object has to be ensured. An artificial cycle is introduced in the EOG by adding a write edge from the modified object (p2 in Figure 3.3) to the artificial object X as well, this cycle will be resolved in the dominator tree phase of the algorithm and the two objects become **peer**. The cycle will also ensure that no argument or return type is annotated with **rep**, because as soon as there is a write operation on one of the parameters, the cycle will be added making the artificial object and the parameter object **peer** to each other.

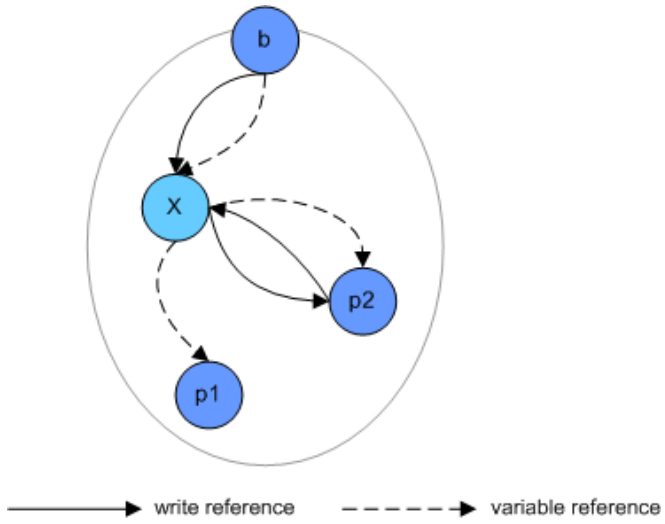

Figure 3.3: Write operation within a static method to a parameter p2. A cycle of write references is inserted between the artificial target object X representing the static method call and the object being written to, p2. This cycle will cause the two objects to be in the same context after all conflicts in the EOG are resolved. The variable references between X and (p1, p2) are added if p1 and p2 are parameters of the static method.

Method Exit The static method exit event is handled like any other method exit event. A variable reference from the artificial object to the return value is added. The return type can be annotated like return types of instance methods. A non-**rep** annotation is ensured as we have shown in the previous paragraph.

Static Initializers

For every class there is one special kind of static method, the *static initializer*. This initializer method is called *clinit* and will be executed as soon as the class is loaded by the classloader. Since these methods are not called from one object to another and do not contain argument and return values, nothing in their signature can be annotated. Therefore, no variables need to be created upon a method entry event caused by a static initializer. We have treated the static initializers to be running in the root context. As soon as a static initializer is called, a new artificial target object is added to the root context. After that, the static initializer is handled just like any other static call.

Object creation

Similar to the invocation of static method calls, the invocation type for object creations needs to be specified as well. The type can be either **peer** or **rep** just like for static method calls, or **peer** x, **rep** x for array types, where x denotes any allowed Ownership modifier. Since object creations are

already being traced by the Tracing agent through the method entry events of the corresponding constructors, we can add a variable reference to the EOG from the caller of the constructor to the created object. This variable reference will then be annotated just like a static method call.

3.3 Implementation

In order to implement the behavior specified in the previous section, we needed to add three new classes to the graph package.

StaticCallGObject The class StaticCallGObject is used for the artificial objects that act as targets of static calls, it is a subtype of GObject. By making it a subtype of GObject and adding it to the graph, it will be treated like any other object when the dominator tree is built up. At the end of the dominator phase it will be either **rep** or **peer** to the caller, as we have shown in Section 3.2. The object also acts as the source for the variable references to the parameter and return objects (in the instance method case, the target object of the call is the source). This allows the algorithm to annotate the parameter and return types like instance method parameter and return types, by examining the relationship between the StaticCallGObject and the parameter/return objects.

StaticCallVariable To ensure that the HarmonizationVisitor annotates static calls, a new type of variable – the StaticCallVariable – is introduced (see Figure 3.4). Objects of this variable type are used by variable references that connect the caller object of a static call with the StaticCallGObject representing the target object. The StaticCallVariable is a subtype of Variable and will be treated like any other Variable by the HarmonizationVisitor. The invocation type of the call is given by the relationship between the source and the target objects of the variable references using a specific StaticCallVariable instance. In order to be able to uniquely identify calls to the same static method (see Listing 3.2), the StaticCallVariable needs to store the called method, the calling method and the bytecode location of the call within the calling method. The bytecode location is necessary, because the same method can be called from different locations within a method (e.g. the calls on line 9 and 11 in Listing 3.2). If there was any loop unrolling when the bytecode was generated from the source, i.e. one call in the source code is mapped to several calls in the bytecode, the tool would not produce correct results. This is why no bytecode optimization should be performed.

Listing 3.2: Static calls are identified by their bytecode location, calling method, and called method. Different calls to the same method (e.g. the calls on line 9 and 11) need to use different target objects, while multiple executions of the same call (e.g. the call within the loop on line 11) need to use the same target object.

```
1   class X{
2       static void doStatic1 (){...}
3       static void doStatic2 (){...}
4   }
5
6   class Y{
7       void callStatics (){
8           peer X.doStatic1 ();
9           rep X.doStatic2 ();
10          for (int i=0; i<10; i++){
11              peer X.doStatic2 ();
12          }
13      }
14  }
```

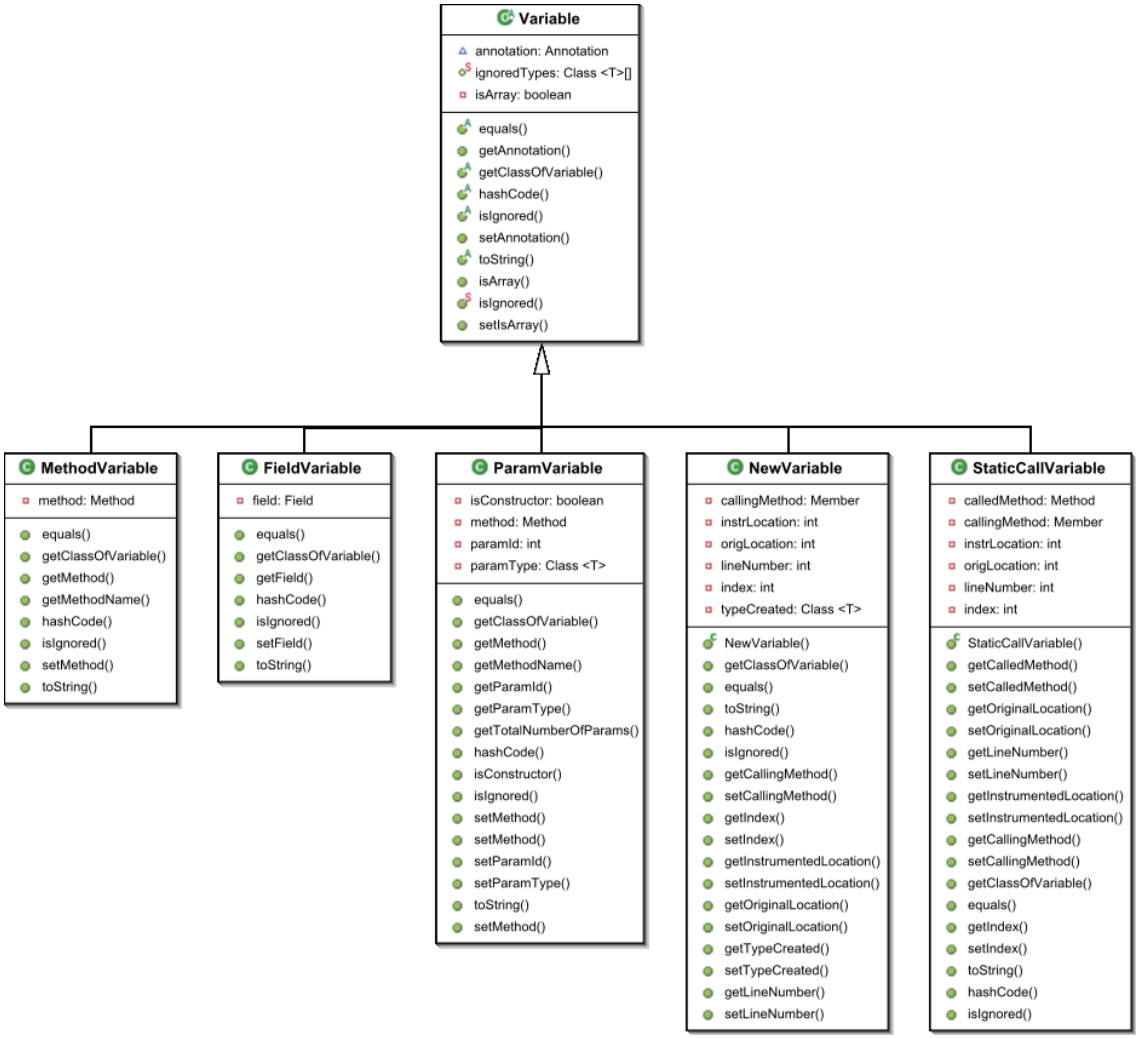

Figure 3.4: UML diagram of the Variables. The StaticCallVariable was added to support the annotation of static calls. NewVariable was added to support the annotation of object creations. The two variable classes store the bytecode location of the call in the original bytecode and the location after the instrumentation. The bytecode location will have to be transformed into an index value during the method body inference phase.

NewVariable The NewVariable is very similar to the StaticCallVariable . The only difference is that the NewVariable stands for an object creation (i.e. a new-statement in the source code) rather than a static call.

Tracing agent changes

The additional information for static method calls, the calling method and the bytecode location of the call within the method is provided by the Tracing agent. The agent can easily gather the bytecode location and the calling method when handling the MethodEntryEvent, since they are passed to the callback function as arguments. The bytecode information needs to be processed by the Type inferer to produce the desired annotation output (which will be described in Section 3.4), this processing is done during the method body inference, since it requires some knowledge of the bytecode of a method under consideration. Method body inference will be described in Chapter 5. Because the trace file is event driven and each event is independent of the others, the

Type inferer needs to be able to ”remember” which StaticCallGObject is currently in charge (i.e. which static method is on top of the call stack), to be able to add the write references to the correct object. For write operations within instance methods, the Tracing agent writes the ID of the object currently in charge to the trace file; for write operations within static methods, this ID is always 0. Therefore, we keep a stack with all the objects representing static method calls (StaticCallGObjects) during the BuildUp phase. The top of the stack can be used as source for write operations on other objects.

Our approach may lead to problems when more than one thread is being executed in the system. If there is only one static call stack maintained and the program switches between two threads, the object on top of the call stack may not be the one that is currently in charge. Suppose thread 1 enters a static method foo(), the corresponding static call object is put on the static call stack. Then thread 2 takes over and enters another static method bar(), whose static call object is also put on the stack. If the program switches back to thread 1 now, the static call object on top of the stack is not the one actually in charge. In order to be thread-save, one static call stack per thread must be maintained.

The thread number of the thread currently being executed is always passed to the callback functions of the Tracing agent and this information could be used to maintain multiple static call stacks (i.e. the thread number is passed for every event). However, we have experienced some problems with this thread ID as it seemed to change suddenly and without cause even in single-threaded applications. This behavior will have to be examined more closely in the future, so the current implementation is limited to single-threaded applications.

3.4 Annotation Output

The annotation output for static calls and object creations are the following: All static calls within a method to the same class are indexed, starting from zero. It is not specified which method was called, the class defining the method is sufficient. For the examples in Listing 3.2, the tool would produce the output in Listing 3.3.

If Marco Meyer's annotation tool [20] is used to insert the annotations to a source file, it is important that static calls include the name of the declaring class (i.e. X.doStatic() instead of just doStatic()). In the Java syntax this is not necessary for a method of the same class, but the annotation tool cannot deal with it otherwise.

Listing 3.3: Annotation output of example 3.2.

```
<ann: static_call  modifier="implicit_peer "  index="0" type="X" />
<ann: static_call  modifier="rep"  index="1" type="X" />
<ann: static_call  modifier="implicit_peer "  index="2" type="X" />
```

The indexing of new-statements is similar, except that all new-statements use the same index, the created class does not matter. Listing 3.5 shows the output for the example in Listing 3.4.

Since the Tracing agent only produces output that identifies different static calls by their bytecode location, the Type inferer needs to translate this information somehow into the indexes required for the annotation output. This can only be done with information of the original bytecode, so this translation is done at a later stage during the annotation of method bodies and is described in Section 5.5.3. A translation only makes sense if the method body inference is activated for a given method anyhow.

Listing 3.4: New-statements within method bodies are numbered from top to bottom.

```
1  class A{
2      void newExample(){
3          peer Object x= new peer Object();
4          rep A a= new rep A();
5          for (int i=0; i<10; i++){
6              peer Object obj= new peer Object();
7          }
8      }
9  }
```

Listing 3.5: Annotation output of example 3.4.

```
<ann:new modifier="peer" index="0" type="java.lang.Object" />
<ann:new modifier="rep" index="1" type="some.package.A" />
<ann:new modifier="peer" index="2" type="java.lang.Object" />
```

Chapter 4

Arrays

The Inference tool that Lyner developed during his master thesis [19] did not handle arrays at all, because it is hard to gather information about the creation, modification, and access of array objects with the JVMTI. Array operations do not trigger the standard events defined by the JVMTI, such as *method entry*, *method exit*, or *field modification*. In this chapter we present a way to handle array operations that relies on the bytecode instrumentation features of the JVMTI.

The notation used in this chapter sticks to the notation of the Java Language Specification [13] where possible. The specification states the following about arrays and their component types:

> All the components of an array have the same type, called the *component type* of the array. If the component type of an array is T, then the type of the array itself is written T[]. The component type of an array may itself be an array type. The components of such an array may contain references to *subarrays*. If, starting from any array type, one considers its component type, and then (if that is also an array type) the component type of that type, and so on, eventually one must reach a component type that is not an array type; this is called the *element type* of the original array, and the components at this level of the data structure are called the *elements* of the original array.

If a new value is assigned to an array component, we will call this a *component update*. If an object that is referenced by an array component is updated, we will call this an *object update using component reference*, i.e. the reference stored in an array component was used to perform the update on an object.

The remainder of this chapter is organized as follows: In Section 4.1 we will show how arrays are handled in the Universe type system and we show why they were not handled in Version 1 of the Inference tool. In Sections 4.2 and 4.3 we discuss all one- and multidimensional array operations and what modifications they entail on the Extended Object Graph. In Section 4.4 we present our implementation using bytecode instrumentation and show what additional steps are needed in the harmonization and annotation phase of the algorithm.

4.1 Arrays in the Universe type system

In the Universe type system, references to arrays of reference type need two Ownership modifiers: the first describes the relationship between the **this** object and the array, the second the relationship between the array and objects referenced by its components. The latter may never be **rep**, because a **rep** annotation would always yield a **readonly** reference when the type combinator is applied with the first Ownership modifier. This would make the references unusable by all objects (even owners of the array). For arrays of primitive type the second Ownership modifier is omitted and the array can be treated like any other object variable. An example EOG with an array variable to be annotated is shown in figure 4.1.

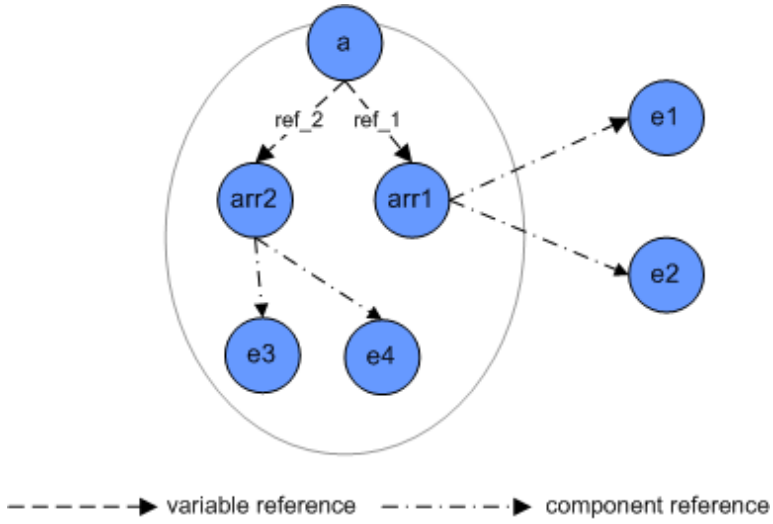

Figure 4.1: Arrays in the Universe type system: ref_2 would be annotated with **rep peer**. The first Ownership modifier describes the relationship between object a and the array, the second modifier the relationship between the array and the objects referenced by its elements. Ref_1 would be annotated **rep readonly**.

The main problem with the array event generation is that array creations (Listing 4.1) and component updates (Listing 4.2) do not trigger any JVMTI callbacks. Only modifications of fields of array type (Listing 4.3) do so. For this reason, arrays were completely ignored in Lyner's work. We had to develop the handling of arrays from scratch and take a look at all array operations and their implications on the EOG.

4.2 One-dimensional Array Operations

For each one-dimensional array operation we discuss what actions need to be taken upon such an operation on the EOG during the build-up phase of the algorithm. We also state what information needs to be provided by the JVMTI to be able to perform these actions on the EOG. At this stage we are not interested how the information is generated. This will be discussed in Section 4.4.

Creation

Array creation (Listing 4.1) is handled like normal object creation. A write reference from the **this** object to the newly created array object is added to the graph. The event generated by the JVMTI needs to provide the references to the creating object and the created array object. It does not matter if an array of reference type or a primitive array was created.

Listing 4.1: Array Creation. Array arr is created as an array of 5 Objects.

```
void create() {
        Object[] arr= new Object[5];
        //do something with arr
}
```

Array Component Update

In order to be able to handle array component updates, we need to introduce a new kind of reference for the EOG. So far we have seen *variable references*, which are used to indicate variable relationships between two objects (e.g. field variables, method return values, etc.), and *write references*, which represent write accesses from one object to another. For arrays we also need *component references* that connect array objects with the objects referenced by its components. A simple variable reference would not be suitable, because there is no corresponding variable. For every variable reference there should be exactly one variable in the source code to be annotated. However, for a component reference of an array, there is no such variable. More precisely, an array component may be used by many variables and its annotation may be different for the various variables using this component.

The example in Figure 4.2 illustrates this problem. There are two objects a and b and two array objects arr1 and arr2. Suppose the algorithm comes to the harmonization and annotation phase with this EOG. First, variable var1 is annotated. The annotation for the arrays that are stored in var1 is **rep peer**, i.e. the component reference ref1 is annotated with **peer**. Now, the algorithm tries to annotate variable var2. There are actually two variable references using var2, so all of them need to be considered. The correct annotation for this variable is **readonly readonly**. The first Ownership needs to be **readonly**, because arr1 is neither in the **peer** nor in the **rep** context of b. The second Ownership modifier needs to be **readonly**, because el2 is not in the **peer** context of arr2.

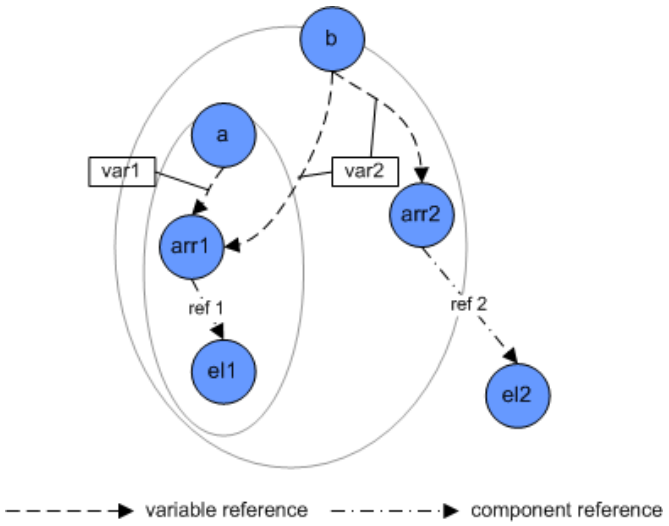

Figure 4.2: Array Component References.

Listing 4.2: Array Component Update. The component of array arr at the given index is updated. After the component update, the array component at the index holds a reference to object e.

```
void setArrayComponent(Object e, int index) {
        this.arr[index] = e;
}
```

For array component updates, a distinction has to be made between the update of a primitive component and an update of a component of reference type. If the array is of primitive type, the component update is treated as a simple write access on the array object. An update of an array component with reference type (Listing 4.2) has the following implications (see Figure 4.3): First, the array object arr will get a component reference to the stored object e. Second, the component

update is treated as a write access on the array object: a write edge between the **this** object (a in this case) and the array arr is inserted.

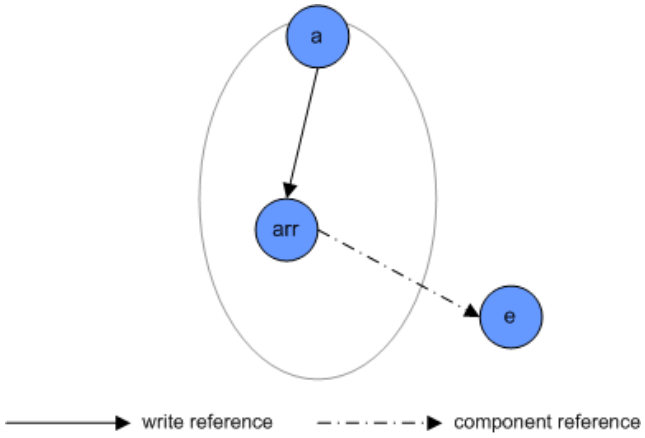

Figure 4.3: Array Component/Element Update. The write reference between a and arr is inserted to indicate that the update is a write operation on the array. The component reference between arr and e is added to indicate that the array now holds a reference to the object e.

The information needed for this manipulation of the EOG is the writing object a, the array object arr and the referenced object e.

Modification of a Field of Array Type

A modification of a field of an array type (Listing 4.3) can be treated like a field modification of any other variable type: a write reference from the **this** object to the modified object x is inserted and a variable reference between object x and the array object arr is introduced. Obviously, the references to the **this** object, the modified object x and the array object arr are needed to apply the changes to the graph. No distinction between arrays of reference type and arrays of primitive type is necessary.

Listing 4.3: Modification of a Field of Array Type. The field array is modified such that a reference to the array arr is now stored in the field.

```
class X{
        Object[] array;
        ...
        void setArray(X x, Object[] arr) {
                x.array = arr;
        }
}
```

Object Update using Component Reference

A write operation (either a non-pure method call or a field modification) performed on an object using a reference stored in an array component (Listing 4.4) is somewhat more complicated to handle. Obviously, this operation can only be performed on arrays of reference type. The algorithm has to ensure that none of the Ownership modifiers of the array field are annotated with **readonly**, because the Universe type combinator applied to the two modifiers may not yield **readonly**. Furthermore, the array object and the object referenced by the array component need

Listing 4.4: Write operation on an object using a reference stored in an array component. The reference stored in the array at a given index (4) is used to perform a write operation (method call doWrite()) on an object. The Tracing agent does not observe that the array variable was used to access the object. Only the write access from **this** to x is observed.

```
X[] arr;

void arrayComponentWrite() {
        X x= new X();
        arr[4] = x;
        this.arr[4].doWrite();
}
```

to be **peer** to each other. This is due to the fact that a **rep** annotation is not allowed there (the result of the type combinator would also yield **readonly**). The following changes must be made to the graph to ensure these properties (Figure 4.4):

- Insert a write reference from the **this** object (a in this case) to the object referenced by the array component el, because the access must be treated as a write operation on the component performed by the **this** object.
- Insert a write reference from a to the array object arr. This ensures a non-**readonly** annotation of the array.
- Insert a cycle of write references between the array object arr and the object referenced by the array component el to ensure a **peer** relationship between the two.

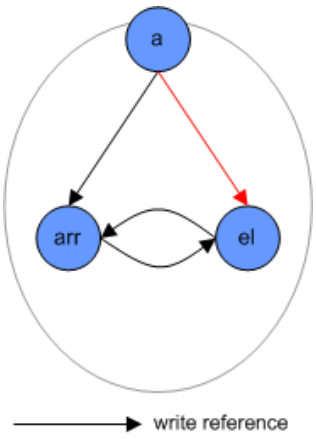

Figure 4.4: Write operation on an object using a reference stored in an array component. Variable references and component references are omitted in the example. The cycle between arr and el is needed to ensure a **peer** relation between them. The write reference from a to arr ensures a non-**readonly** annotation of the array. However, the only reference that can actually be added is the red one from a to el, due to limitations of the JVMTI.

The Inference tool needs to know the source and target object of the write operation (a and el) as well as the array object arr that was used to get the reference to the object el.

Since we did not find a way to generate all of this information with the JVMTI, we had to come up with a different solution. The problem is that we cannot distinguish if the object el was accessed through a direct reference or through a reference stored in a component of the array

arr (see Listing 4.5). The only information that can be extracted from the JVM are the writing object a and the updated object el. This problem is similar to the "dereferencing chains" problem described by Lyner in Section 4.3.4 of his Master Thesis [19], so we have no choice but to treat this operation like a direct write access on object el. This means that the write references to and from the array object arr will *not* be added to the graph, therefore a non-**readonly** annotation of the two Universe type modifiers is not guaranteed. However, as we will show in Chapter 5, the write operations using a **readonly** annotated array can be made legal by casting the reference component to the correct Universe type during the annotation of the method bodies.

Listing 4.5: The JVMTI cannot distinguish between the two write operations on lines 6 and 7. There is no way to figure out whether the write operation was performed using the array variable arr or the Object variable x.

```
1  Object[] arr;
2  ...
3  Object x= new Object();
4  arr[2]= x;
5  ....
6  arr[2].doWrite();
7  x.doWrite();
```

4.3 Multidimensional Array Operations

Multidimensional arrays introduce even more difficulty, because of the way they are implemented in Java. They are implemented as an array that stores references to other arrays (we will call them *subarrays*). Since multidimensional arrays are also annotated with only two type modifiers, the subarrays that form the multidimensional array all have to be in the same context. Lyner proposed the following solution to this problem: whenever a one-dimensional array aone is added to a multidimensional array amult, the write references aone–amult and amult–aone are inserted in the EOG (see Figure 4.5). By creating this cycle, the conflict resolution step will make both objects **peer** to each other. Since there is no variable reference between the two, the reference will not be annotated. This is the easiest way to make sure that all arrays making up a multidimensional array end up in the same context, so we have chosen to implement it this way.

There are some more differences between the one-dimensional array operations and the multidimensional ones. Next, we take a look at the different operations on multidimensional arrays.

Creation The new-statement for multidimensional arrays initializes the array object and a given number of subarray dimensions. If the array has m dimensions, the programmer can specify the sizes of the dimensions 1 to n, leaving the sizes of dimensions n+1 to m open. This means that all the subarray objects until dimension n are also created and initialized, while the rest are set to **null**. The elements, objects in the highest dimension, are always initialized to **null**. Upon the creation of a multidimensional array, a write reference is added to the multidimensional array amult (Figure 4.5). Additionally, write references from amult to the one-dimensional subarrays (aone, atwo) and back are needed to introduce the cycle of write references described at the beginning of this section.

Array Component Update For multidimensional arrays there are two kinds of component update possibilities. Either one of the references to the subarrays that make up the multidimensional array is updated or an actual element, i.e. a component that holds a reference which is not a subarray, is updated.

The update of an element reference is illustrated by Listing 4.6. This operation can be handled exactly like a simple array component update for one-dimensional arrays of reference type. A write reference from the **this** object to the subarray aone (ref 2 in Figure 4.5) is added and a component

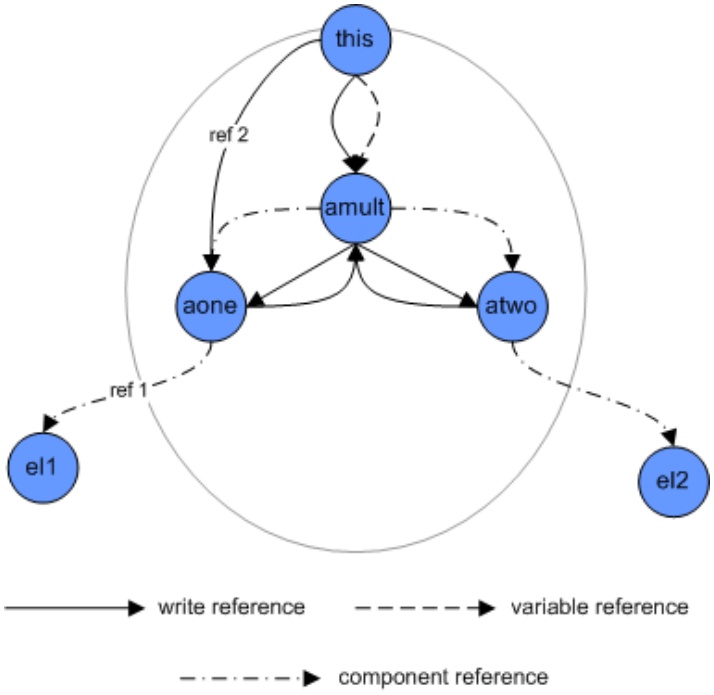

Figure 4.5: EOG of a two dimensional array amult with subarrays aone and atwo. The subarrays always need to be **peer** to the multidimensional array. This is ensured by the cycle of write references between them. A write reference to any of the arrays (e.g. ref2) has the same effect as a write reference to any other array, since they will all stay in a **peer** relationship.

reference to the object el1 referenced by the element of the subarray aone (ref 1) is inserted. A non-**readonly** annotation of the multidimensional array amult is guaranteed even though there is no write reference from **this** to amult added. This is due to the fact that the subarrays and the multidimensional array are **peer** to each other (due to the write cycle that was created between amult and its subarrays). Because of this **peer** relationship, a write reference to any of the subarray objects has the same effect as a write reference to amult. The information needed is therefore the same as for a one-dimensional array component update: the writing object **this**, the array object aone and the object el1 referenced by the array element. Since the handling of this event is exactly the same as if the subarray was used as a one-dimensional array, the algorithm does not need to distinguish these two cases.

Listing 4.6: Multidimensional Component Update

```
Object [][]  amult= new Object[2][10];
Object el1= new Object();
...
amult[0][1]= el1;
```

When a component holding a reference to a subarray is updated (see Listing 4.7), a write reference cycle between amult and aone needs to be added to the EOG to ensure a **peer** relationship between the multidimensional array and its subarrays. Additionally, a write reference between the **this** object and amult is added to indicate that a write operation on the array object was performed. The information needed in this case is amult, aone and **this**, just like for the case of an update of an element described earlier.

The two cases of multidimensional array component updates (subarray update and element

update) need to be distinguished. In the case of an update of a reference to one of the subarrays, a write cycle has to be inserted, in the case of an update of reference to an actual element, no write cycle needs to be added. The component reference between the array and the object referenced by the component must be added in either case.

Listing 4.7: Component with Subarray Reference Update

```
Object [][]  amult= new Object[2][10];
Object []  aone= new Object[10];
 ...
amult[0]= aone;
```

Since every array is a subtype of java.lang.Object, we run into problems when arrays of type Object[] are used to store references to other arrays (see Listing 4.8). In this case, our tool will make array b **peer** to array a, because it thinks that b is a subarray making up the (allegedly) multidimensional array a. Checking if the signature of a is multidimensional does not work either, because Object [][] arrays can be used to store array references in their elements as well.

Listing 4.8: Component with Subarray Reference Update

```
Object []  a,  b;
a= new Object[10];
b= new Object[10];
 ...
a[2]= b;
```

Modification of a Field of Multidimensional Array Type A simple assignment of a multidimensional array object to a field can be handled like the one-dimensional case. A variable reference to the assigned multidimensional array object is added to the owner of the field.

Object Update using Component Reference The same problems occur as for one-dimensional arrays. Conceptually, the write operation on an object referenced by a multidimensional array element (Listing 4.9) can be treated like a write operation in the one-dimensional case. A write reference from the subarray aone to the element being written to, el1 and a write reference from the modifying object (**this**) to the subarray aone are sufficient to establish the correct relationship between **this** and amult for the same reason as explained for the component update (due to the **peer** relationship of all subarrays).

As we have shown in the section about one-dimensional arrays, it is not possible to add the write references to the array object, so the only write reference added is the one from the writing object, **this** in this case, to object el1 referenced by the array element.

Listing 4.9: Write operation on a multidimensional array element.

```
SomeClass [][]  amult;
 .....
amult [0][1].  field = value;
```

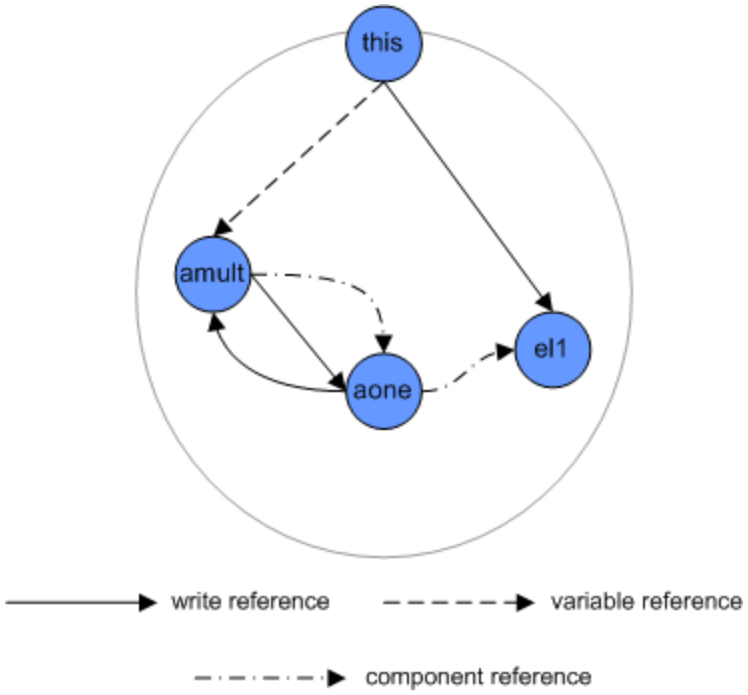

Figure 4.6: Object Update using Component Reference. Object el1 is updated using a reference stored in the subarray aone of amult. The only write reference that can be added is the one between **this** and el1.

4.4 Implementation

4.4.1 Bytecode Instrumentation

As we pointed out earlier, not all of the events and information mentioned in the previous section can be generated with the standard JVMTI callbacks. It is possible to register callbacks for events such as field update/modification, method entry and method exit. The field modification callback can be used to cover array field updates, so nothing special has to be done for this operation. However, array operations such as component updates or array creations do not trigger any callbacks.

The JVMTI provides a callback that is triggered whenever a new class is loaded by the JVM. This callback can be used to instrument the bytecode of the class, i.e. injecting bytecode instructions that can alter the operand stack or call methods. We use this possibility of bytecode instrumentation to generate the missing events for array operations. The goal of the instrumentation is to call a specific method of the Type inferer with the necessary references extracted from the JVM stack passed as arguments. The bodies of the called methods are empty, but the calls will trigger a MethodEntryEvent in the Tracing agent which is used to generate two custom events: ArrayCreatedEvents and ArrayCompUpdateEvents (for component updates).

In the future, the Tracing agent might be incorporated in the Type inferer, so that the time consuming indirection via xml trace file can be omitted. The tracing would then be done with the Java Debug Interface (JDI), which unfortunately does not provide facilities to do bytecode instrumentation at runtime. We think it would still be possible to use the bytecode instrumentation features of the java.lang.reflect package to achieve the same results as with the JVMTI. However, the advantage using trace files is that multiple test runs can be joined and therefore a better code coverage may be achieved.

The instrumentation of the bytecode uses some operand stack manipulation instructions, Table 4.1 shows their definitions.

Instruction	Effect
dup_x1	Duplicate the top word of the stack to place 3 of the stack
dup_x2	Duplicate the top word to place 4
dup2_x1	Duplicate the top two words to places 4 and 5
dup2_x2	Duplicate the top two words to places 5 and 6
pop	Pop the top word
swap	Swap the top two words
invokestatic	Invoke a static method

Table 4.1: Definitions of the bytecode instructions used to instrument the array instructions. The word *stack* denotes the operand stack of the currently executing method. The numbering of the places on the stack starts with one.

anewarray and newarray instrumentation

The two opcodes for one-dimensional array creation, anewarray for arrays of reference type and newarray for arrays of primitive type can be instrumented the same way, because these creation events are both handled just like normal object creations. The instrumentation is only needed because the JVMTI does not generate these events by default. The instrumentation is rather easy: a simple call to the static method arrayCreated(Object array) has to be made, after the newly created array object was duplicated on the stack. The tracing tool creates an ArrayCreatedEvent upon a method entry event to this specific method.

multianewarray instrumentation

The multianewarray instruction is used to create both arrays of reference type and arrays of primitive type with more than one dimension. The JVM stack before and after this operation can be seen in Table 4.2. The multianewarray instruction initializes the multidimensional array object and a given number of subarray dimensions, as described in Section 4.3. The number of subarray dimensions to be initialized is part of the opcode for the instruction and is not passed on the stack. The total number of dimensions is given by the type of the array, which is stored in a constant pool entry. The instrumentation code consists of a duplication of the array object followed by a call to the static method multidimArrayCreated(Object array). The Tracing agent will then be able to get the references of all subarrays; they are collected by a recursive traversal through the objects referenced by the components that proceeds with the next object as soon as a **null** reference was found. Anything but a **null** reference in a subarray component must be a subarray again because only the subarrays are initialized after the multianewarray instruction.

Before	After
size of dimension n	
...	
size of dimension 2	
size of dimension 1	array reference
...	...

Table 4.2: JVM stack before and after a multianewarray operation. The multianewarray instruction takes the number of dimensions to be initialized (n) as an argument. The total number of dimensions is given by the type of the created array.

Xastore instrumentation

The family of array store instructions of primitive type bastore, castore, dastore, fastore, iastore, lastore, sastore are used to store a component of a primitive type in a given array. These bytecode instructions correspond directly to the array component update operation described in Section

4.2. For primitive arrays it does not matter what is stored in the array, because this event will be handled as a simple write access to the array object. The operand stack for array store instructions with types that are one word wide can be seen in Table 4.3, for two word wide types refer to Table 4.4. The goal of the instrumentation is to duplicate the array object and call the static method primitiveArrayCompUpdate(Object array). Table 4.5 shows an instruction series that achieves this goal. In Table 4.6, the effect on the stack of each instruction within the series is illustrated. The sequence for double word types is slightly different, but basically it does the same thing. The MethodEntryEvent triggered by this method call will be used to generate a ArrayCompUpdateEvent.

Before	After
value	
index	
array	
...	...

Table 4.3: JVM stack before and after an Xastore instruction (where 'X' denotes a single word primitive type).

Before	After
value word1	
value word2	
index	
array	
...	...

Table 4.4: JVM stack before and after an Xastore instruction (where 'X' denotes a double word primitive type).

Initial bytecode	Instrumented bytecode
...	...
bastore	dup2_x1
...	pop
	pop
	dup_x2
	invokestatic
	bastore
	...

Table 4.5: Initial bytecode and the instrumented bytecode for a bastore instruction.

	Initial stack	dup2_x1	pop	pop	dup_x2	invokestatic
Stack	value	value				
	index	index	index		array	
	array	array	array	array	value	value
	...	value	value	value	index	index
		index	index	index	array	array
		...	...	...	...	...

Table 4.6: Bytecode instruction series that will produce the desired stack before the call to the static method primitiveArrayCompUpdated(Object array).

aastore instrumentation

The aastore instruction is used to store a reference at a given component of an array of reference type. This can be either a subarray or an object of an arbitrary reference type. Table 4.7 shows how the JVM stack looks before and after an aastore instruction. The goal of the instrumentation is to duplicate the value and the array references and to have them at the top of the stack before calling the static method objectArrayCompUpdate(Object[] array, Object value) with these two arguments. Table 4.8 shows what the operand stack looks like before the instrumentation code is executed and what it should look like before the static method is invoked. A series of bytecode

instructions showing a step-by-step transition from the original stack to the stack before calling the static method is listed in Table 4.9. The static method objectArrayCompUpdate() will then produce a MethodEntryEvent that will be used by the Tracing agent to produce an ArrayCompUpdateEvent.

Before	After
value	
index	
array	
...	...

Table 4.7: JVM stack before and after an aastore instruction.

T1	T2
	value
	array
value	value
index	index
array	array
...	...

Table 4.8: JVM stack before the instrumentation instructions are executed (T1) and before the static method is invoked (T2).

	Initial stack	dup_x2	pop	dup2_x1	pop	swap	dup_x1
				index			value
		value		array	array	value	array
Stack	value	index	index	value	value	array	value
	index	array	array	index	index	index	index
	array	value	value	array	array	array	array
	...	...	...	...	...	...	...

Table 4.9: Bytecode instruction series that will produce the desired stack in Table 4.8. The invokestatic instruction that actually calls the method objectArrayCompUpdate() is omitted here.

Bytecode Engineering Tool

In order to manipulate the bytecode of the loaded classes we evaluated several bytecode engineering tools. The main requirement for a tool was that it is easily usable from the C++–environment of the JVMTI agent, so we first searched for a library that is written in C/C++. However, apart from the *java_crw_demo* that comes with the Java Development Kit there are no such libraries. The java_crw_demo is very hard to use and a lot of additional coding would have been necessary to make it powerful enough for our purpose. Therefore we have chosen to take a look at the various Java bytecode engineering libraries and invoke them from the JVMTI agent using the JNI, the Java Native Interface.

jclasslib is a library that was developed for a bytecode viewer program. It is very versatile, but it has too many unnecessary features that are used for the viewer program and are not needed for our purpose.

BCEL is the most often used bytecode engineering library and it is part of the Apache Jakarta project. It provides the possibility to parse and manipulate Java class files. However, the JVMTI callback gets a byte array representation of the loaded class and we did not find a simple way to parse this byte array into the BCEL internal data structure.

javasist is a high-level bytecode manipulation library that allows to conveniently inject Java bytecode to method bodies and constant pool entries. It offers the possibility to inject bytecode in the form of source text that is compiled into bytecode and then injected.

ASM is, according to their analysis [10], the library with the best performance. It allows dealing directly with a class constant pool and offsets within method bytecode. Much of its implementation is based on the Visitor pattern. It has a lot of potential, but it would have needed a lot of time to get familiar with the way it works, so it was not chosen to be used in this project.

Javasist [3] turned out to be the most convenient library to use, because it offers the easiest way to iterate through the bytecode instructions of methods and inject code. Since all libraries are generally very similar, we chose to work with javassist. However, as we will show in Chapter 5, we ended up using BCEL to implement our abstract interpretation implementation for method body inference. The two libraries will never be loaded at the same time, because one is used by the Tracing agent, the other by the Type inferer, so there is no memory overhead when using both libraries. However, it would be nice to use just one tool for both tasks in the future.

4.4.2 Annotation and Harmonization

Only a few changes have to be applied to the harmonization and annotation phase of the algorithm. Variables for fields of arrays of primitive type can be handled like other field variables, they do not have to be treated specially. However, arrays of reference type need to be treated specially (see Figure 4.7). When an array variable is annotated, two Universe types have to be found. The first one describes the relationship between the owner of the variable and the array object (in this case `a` and `arr`). This works the same way as the annotation of normal variables. The second Universe type is found by following all component references of the array (`ref2` and `ref3`). The end of such a component reference may be an array itself (`sub1` and `sub2`), because the original array may be multidimensional. Therefore, the algorithm needs to traverse all subarrays recursively until it reaches objects that are not arrays anymore (`el1` and `el2`), i.e. that have no more outgoing component references. Then the context of the array is compared to the context of those elements and the Universe type is set correspondingly. In the example, a **readonly** annotation would be correct.

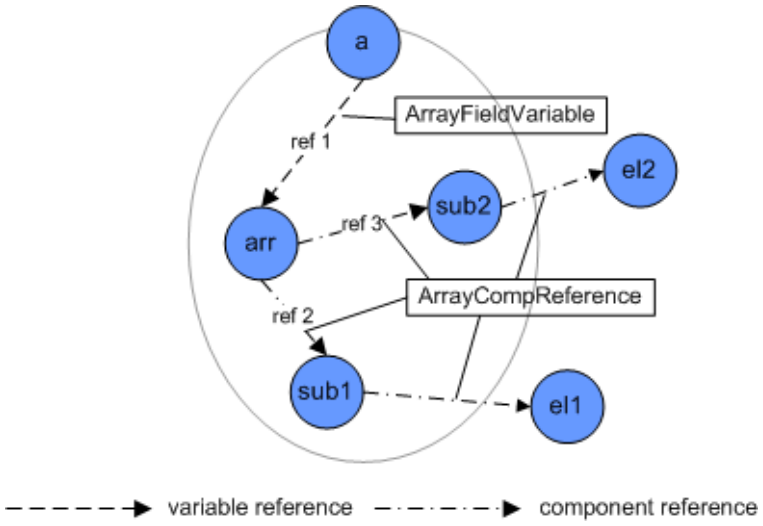

Figure 4.7: References considered during the harmonization and annotation phase.

The fact that we cannot distinguish whether an object was accessed through a reference stored in an array or whether it was accessed directly (as we presented in Section 4.2) when performing a write operation, needs to be considered as well in this phase. Due to the omission of write references from the array object to the object being written to and vice versa, it is possible that

the annotation of one of the array's Universe types may be annotated **readonly** instead of **peer** or **rep**. A scenario that leads to a wrong annotation is illustrated by Listing 4.10 and Figure 4.8. Even though a1 performs a write operation on an object referenced by the array arr (on line 16 of the example code), there is no write reference to the array added. As a result, arr is neither in the **peer** context nor in the **rep** context of a1 and must be annotated **readonly**. It is not clear how we should handle this case. In Chapter 5, we will show how to add casts to the source code. This way we can make the write access on the object referenced by the array compilable, even though the array is annotated with **readonly**. In future versions of the tool, we hope that the user is able to interact with the algorithm and to resolve such conflicts.

There may also be the case that some objects referenced by array components end up in an arbitrary context, because there was not a write access on all objects referenced by the array during the execution of the program. Here again, a **readonly** annotation is found instead of a **peer** or **rep** one. It would be useful as well if the user could interact with the program and specify if all these object should be made **peer**. However, this is rather a problem of code coverage and should be taken care of by using good test cases.

Listing 4.10: Scenario where the "write on array components" simplification leads to a wrong annotation. Object b creates objects arr and a1, which performs write operations on the components of the array. Since we cannot add a write reference from a1 to arr, the array and its components end up in a different context. Only a cast for the write operation on line 16 can make this code compilable.

```
1  class A{
2          public readonly readonly Object[] arr; //arr is wrongly annotated with readonly readonly
3
4          public A(readonly readonly Object[] arr){
5                  this.arr= arr;
6          }
7
8          //Makes another object peer to this one.
9          public void makePeer(peer Object other){
10                 other.writeOp();
11         }
12
13         //execute a write operation on all components
14         public doWriteOnComponents(){
15                 for (int i=0; i<arr.length; i++){
16                         arr[i].writeOp(); //this operation should make the array non-readonly
17                 }
18         }
19
20         //Symbolical write operation (non-pure method call)
21         public void writeOp(){
22         }
23 }
24
25         //somewhere in b:
26         rep readonly Object[] arr= new rep readonly Object[2];
27         peer A a1= new peer A(arr);
28          ....
29         arr[0]= e1;                 //e1 and e2 were created by some other object
30         arr[1]= e2;
31         a1.makePeer();
32         a1.doWriteOnComponents();
```

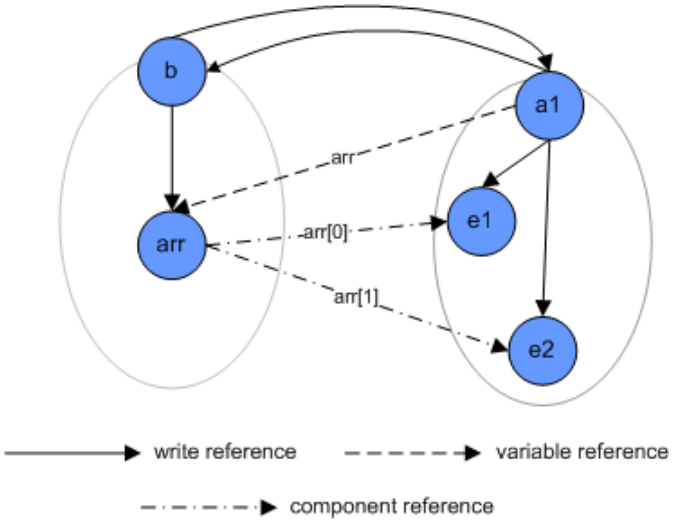

Figure 4.8: Object Graph of the example in Listing 4.10

In order to be able to distinguish between normal variables and array variables (of reference type), a flag was added to the class `Variable`. This flag is set in the build-up phase and checked before a variable is annotated in the harmonization and annotation phase of the algorithm. If it is set, the algorithm will follow the component references of the arrays stored in the variable to find the annotation of the second Ownership modifier.

Chapter 5

Annotation of Method Bodies

The ultimate goal of the Inference tool is the production of annotation output that can be used to generate compilable, Universe typed Java code. So far, we have shown how to annotate fields, method signatures, static method calls, and object creations using the runtime information generated by the JVMTI tracing tool. However, the annotation of the whole method body, e.g. local variables and Ownership casts was not discussed. In this chapter we show why method body annotation was not implemented in Version 1 of the Inference tool and we present a way to annotate method bodies using abstract interpretation.

The remainder of this chapter is organized as follows: Section 5.1 provides an overview of how methods are executed in the Java Virtual Machine and how they are stored in the bytecode. In Section 5.2 we show what is necessary for method body annotation and we point out some difficulties that have to be dealt with. In Section 5.3 we show how we could annotate local variables using a bytecode instrumentation approach and why we did not do so. Section 5.4 focuses on our solution that is based on abstract interpretation. Implementation details are presented in Section 5.5.

5.1 Methods in Java

In order to understand this chapter we present the basic functionalities of methods in Java. Every time a method is invoked in Java, the Virtual Machine creates a new *frame*. The frame is destroyed when the method invocation is completed. "A frame is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions." (JVM Spec, Section 3.6 [25]). Each frame has an *array of local variables* and an *operand stack* as well as a reference to the *runtime constant pool* of the class of the current method. The runtime constant pool of the class contains several kinds of constants, these can be method and field references, string constants, numerical constants and others.

Local Variable Array The array that contains all local variables of a method is called *local variable array* in the JVM Specification. This array is not only used to store all the values of the local variables, but the parameter values of the method are also passed in that array. Non-static methods get a reference to the **this** object at index 0 of the array and the parameters 1 to n at indexes 1 to n. Static methods cannot receive a **this** reference, therefore parameter i is found at index $i - 1$. Since the components of the local variable array are also used to pass the input parameters, we will refer to them as *registers*, as this seems to be the standard in most technical papers.

The JVM Specification does not pose any restriction on the reuse of registers, i.e. one register may be used for an arbitrary number of local variables in the source code and vice versa. The only restriction is that whenever a value is loaded from a register it must be initialized and have a type appropriate to the instruction.

Operand Stack The operand stack is a regular last-in-first-out data structure. Whenever the JVM needs to store an intermediate result it will push it on the stack. The next instruction can then use the previously calculated value by popping it off the operand stack and can push back the result if there is one. Operations are not performed directly on the registers; values in the registers are always loaded on the stack first and then processed. The maximum stack height is given at compile time and must never be exceeded.

5.2 Method Bodies

Listing 5.1 illustrates the different possibilities of Universe type annotation within a method body. For local variables, an Ownership modifier is specified in the *variable declaration* (lines 3 and 4). *Object creations* can either create **peer** or **rep** objects (lines 5 and 7). *Ownership casts* are possible just like normal type casts (line 6). On line eight, a *static method* is invoked. As we have seen in Chapter 3, static calls can run in either the **peer** or **rep** context of the caller and this can be specified by the respective keyword. There, we have already shown how to annotate object creations and static method invocations. Ownership casts and local variable declarations are handled in this chapter.

Listing 5.1: The various possibilities for annotations within a method body.

```
1  class A{
2          rep Object newObject(){
3                  readonly Object ro_local ;                  //local variable declaration
4                  rep Object rep_local ;
5                  ro_local = new rep Object();                //object creation
6                  rep_local = (rep Object) ro_local ;         //Ownership cast
7                  ro_local = new peer Object();    //this causes the readonly annotation of ro_local
8                  peer A. staticCall ();                      //static method invocation
9                  return rep_local ;
10         }
11
12         static void staticCall (){
13                 ...
14         }
15 }
```

There are three cases where Ownership casts are necessary: *local variable harmonization*, *dereferencing chains*, and *code coverage* issues.

At this stage of the algorithm, we decided that we do not alter the annotations that were inferred by the harmonization visitor anymore, i.e. no harmonization of variables is conducted. We did this to ensure the modularity of the different steps of the algorithm. This may lead to the situation that is illustrated by Listing 5.1. Suppose that one of the two new-statements on lines 5 and 7 is annotated with **rep**, the other with **peer** after the harmonization and annotation phase. Each of these created objects is assigned to the same local variable ro_local . If these variables were fields or parameters, the harmonization visitor would have to harmonize the variables and make them all **peer**. However, since the harmonization visitor does not annotate local variables, we have no choice but to assign ro_local the smallest common supertype of all types that were assigned to this variable (**readonly** Object in this case). Furthermore, since the local variable will be annotated with a **readonly** Universe type, the assignment source on line 6 needs to be casted to **rep**.

Another problem was already described by Lyner[19]; it is being referred to as the *dereferencing chains* problem. A dereferencing chain is a number of read accesses followed by a write access (Listing 5.2). Dereferencing chains are problematic, because the Tracing agent only generates an event for the write access to the object that is on the stack at last. Universe modifiers within the chain are not considered, even though the type combinator would have to be applied subsequently after each object is loaded on the stack. In the example, the chain is treated as a write operation from an instance of class A to the object referenced by field c of the object stored in b. The

object referenced by field b is totally unaffected and a **readonly** annotation is possible. However, the type combinator has to be applied for the field access b.c (**readonly*****peer**). The result is a **readonly** reference, making the non-pure method call doWrite() on this reference illegal. A cast to **peer** needs to be inserted. Since there is a write reference between the two objects, we know that they must be either in a **peer** or a **rep** relationship to each other (write references cannot cross context boundaries). However, we do not know which cast is correct, so we have no choice but to chose a default **peer** cast and output a warning. The user can supply another value for the cast in the default annotations input file that can be used instead.

As mentioned in Chapter 4, write accesses to objects referenced by array components pose the same problem. In the example in Listing 5.2, instead of the chain b.c, an array access, such as arr[n], could be substituted where arr denotes an array and n is an index into the array. This array access would result in the exact same situation as the case mentioned in the previous paragraph. Here, a cast will be needed as well.

Listing 5.2: Dereferencing chain. The type combinator returns **readonly** for the read access b.c. However, read accesses do not trigger any event, so the Tracing agent treats the write access on c as a direct write access.

```
class A{                                        class B{
        readonly B b;                                   peer C c;
                                                }
        void foo(){
                ((peer C)b.c).doWrite();        //write operation on c needs cast
        }
}
```

Another case where casts may be necessary are code coverage issues. Remember that until now, we have inferred the Universe types of a program based on a given test run. This means that we may have execution paths that might not have been taken by the program, leading to wrong annotations. Consider the example in Listing 5.3: there are two execution paths; depending on the boolean condition cond, a write operation on obj is performed or not. If by chance, in a given test case, cond was always true, our algorithm will find a **readonly** annotation for variable obj. However, if we add this annotation to the source code and try to compile it, we will get an error, because the write access on line 5 is illegal. It can be made compilable by adding a cast or setting the Universe type of the variable to **peer**. Just like adding casts for dereferencing chains, it is not sure whether a **peer** or **rep** cast is correct, so the user can also supply a default value. As pointed out in the first example, changing the Universe type of a variable at this stage of the algorithm is very difficult, because it may introduce a lot of conflicts and it has an effect on other variables as well. All method bodies accessing the changed variable would have to be revisited again.

As this problem originates from bad code coverage, it is more helpful to achieve better code coverage than to provide workarounds. One improvement is the possibility to join several test cases into one Runtime Inference run. This would obviously increase the code coverage and will be described in Chapter 6.

Listing 5.3: Code Coverage issue.

```
1 readonly T obj;
2 if (cond){
3         ....      //no access on obj
4 } else{
5         ((peer T) obj).doWrite();
6 }
```

5.3 Bytecode Instrumentation

The first idea to annotate the local variables was to use the same approach as for arrays, namely bytecode instrumentation. By instrumenting the astore bytecode, which stores a reference into a register, it would be possible to track which references are stored in the registers and the registers' Universe types could be annotated that way. However, we had some concerns that the tracing of all astore instructions would generate a significantly larger trace file which is already the bottleneck of our Inference tool.

To illustrate this, we have run three tests with the LinkedList example of [19]. The test case inserts about 1000 objects into the list, the linked list's insert method is implemented recursively and has an O(n) runtime. This causes many method entry and exit events to be created. The tests were run on a 900 MHz Pentium III Laptop computer, so disk-I/O was rather slow.

First, we ran the original test program without the Tracing agent enabled. After that, we have run the agent without output enabled, i.e. the events were generated but not written to a file. The last run included writing the output to a file, so the I/O was fully enabled. We have measured the execution time with the linux command time. The results can be seen in Table 5.1. The generated output file was about 115 MB in size and 593'336 events were generated. The example illustrates that the Tracing agent itself produces quite a lot of overhead, and the disk-I/O more than doubles this overhead again.

	Original	Agent (no I/O)	Agent (with I/O)
real	0m0.249s	6m50.240s	24m20.673s
user	0m0.148s	2m51.680s	5m9.395s
sys	0m0.034s	3m31.720s	5m46.682s

Table 5.1: Three test runs with the Tracing agent. The first column shows the original program without the agent. The second column shows the execution time with the tracing agent, but no output is written to a file. The third column shows the execution time including output generation.

The Java part of the program's execution time for this test run is listed in Table 5.2. It illustrates clearly that building up the EOG, which includes the parsing of the input file, requires by far the most time. We also extracted the time the parsing took on its own, without actually building up the EOG data structure. The parsing of all events took 497s, which is more than 50% of the build-up time.

Algorithm Step	Execution Time
Build EOG	952.84s
Dominator	3.16s
Store Dominator level	0.978s
Resolve Conflicts	0.966s
Harmonize and Annotate	19.312s
Method Body Annotation	4.393s
Output	0.509s

Table 5.2: Execution time of each step of the Inference algorithm. It can be seen that building up the EOG takes by far the most time.

The second problem with this approach is that there is no possibility to add Ownership casts. This is because dereferencing chains cannot be detected for the same reason as for other variables. It can only be traced that a write operation on an object is performed, but not how the object got on the stack, i.e. which variable(s) were used to load the object on the stack. In order to get this information every bytecode instruction that puts an object on the stack or removes an object from the stack would need to be instrumented and the trace file would become much too big.

Unless there is a more efficient way to store and load generated events, it makes no sense to enlarge the trace file even more. In general, we can say that it is responsible for about 50% of the execution time.

5.4 Abstract Interpretation

Our approach to annotate method bodies relies on a combination of runtime and static inference. The information gathered during the runtime inference about the method signatures, fields, object creations, and static method invocations will be used while conducting an abstract interpretation of the types on the operand stacks of the methods to be annotated. This approach is static in the sense that not an actual method execution is interpreted, but the bytecode of a method is interpreted abstractly.

The abstract interpretation we need to perform is very similar to the one bytecode verifiers for the JVM implement. However, in our case the bytecode is not verified, it is assumed to be correct and well-typed. This means that all checks of the stack-size and Java type checks can be omitted. At this point, we have annotated the class fields, method signatures, static method calls, and object creations using the information gathered by the Tracing agent running a test case on the given program. Unless the code coverage of the given test case is 100%, there might be some variables without inferred Universe type, because no operation using these variables was traced. In a first step, we will simply use a default Universe type for those variables. If the default value does not produce a correct result, the user may specify a Universe type that is passed in the default annotations input file.

Since this stage of the algorithm will not deal with runtime information anymore, it has to take the Java bytecode (either in a .jar or .class file) as input. The bytecode and additional information of a class' methods are stored in the classfiles in so-called *method_info* structures, which further contain *attribute_info* structures. The most important attribute_info of the method_info structure is the *code_attribute*, it contains the bytecode of the method. Depending on the arguments supplied to the javac compiler, a LocalVariableTable and/or a LineNumberTable are generated as attributes of the code_attribute. By default, only the LineNumberTable is generated. The LocalVariableTable is generated with the argument **-g**. The LocalVariableTable is especially interesting for our purpose, because it contains information on how the local variables of the sourcecode are mapped into the local variable array of the method in the bytecode. Without this information we have no information on the variable names, scope and Java type, therefore we require the table to be present for the method bodies that need to be annotated. The LineNumberTable contains information about the mapping between the bytecode location of an instruction and the location of its pendant in the source code. Since we need to know the ordering of the instructions in the source code and since there is no way to generate this information from the raw bytecode, we require a LineNumberTable as well. The ordering of bytecode instructions within a line of source code can still not be determined, which could yield wrong annotations if there are several statements on one line. This can only be solved by spreading multiple statements over multiple lines. An example where statements on the same line of source code appear in another order in the bytecode is shown in Figure 5.1. Two new-statements and their corresponding constructor calls are nested, so the order they appear in the source code is not the same as the order they appear in the bytecode.

5.4.1 Algorithm Overview

Our abstract interpretation algorithm starts building up a call-graph first. The call-graph keeps track of all methods that are called within the body of a given method. The call-graph is necessary, because, as it was pointed out earlier, there might be some variables with an unknown Universe type due to bad code coverage. These unknown types (e.g. method parameters/return types) may become known during a method body inference, so the signature of a method can change. A signature change of a method may have an impact on the method body inference of all its callers, so they have to be revisited. The algorithm therefore runs a fixpoint iteration over all

```
public void newTest(){
        new Integer(new Short((short)5));
}
```

```
 0: new #34 <java/lang/Integer>
 3: new #36 <java/lang/Short>
 6: dup
 7: iconst_5
 8: invokespecial  #39 <java/lang/Short.<init>>
11: invokevirtual  #43 <java/lang/Short.shortValue>
14: invokespecial  #46 <java/lang/Integer.<init>>
17: return
```

Figure 5.1: Source- and bytecode of the same method. The new-statements and their corresponding constructor calls are nested in the bytecode. In general, it is not specified in what order statements appear in the bytecode. This means that our tool may run into problems when there are multiple statements on one line of source code.

methods that are supposed to be annotated. The method body inference should be conducted in an optimal order such that the least amount of methods need to be revisited. David Graf has implemented an algorithm by Sălcianu and Rinard [24] finding such an optimal order in his work [14]. It tries to visit methods first which others are dependent on. In our current implementation, we run the method body inference in a random order. The performance should still not be much worse, because if the test cases are carefully chosen (i.e. code coverage is high), method signatures change very seldom during method body inference.

The user may specify which classes or packages should be considered when the method body inference is conducted. It does not make sense, for example, to run a method body inference on library classes for which there is no sourcecode to be annotated. For these classes, only the signatures of the methods are interesting. The classes or packages to be annotated are specified in the configuration file supplied to the Type inferer.

In order to conduct the method body inference of a single method, we use an adapted version of the bytecode verifier algorithm described in Chapter 4.9.2 of the Java Virtual Machine Specification [25]. The bytecode verifier is used to statically check bytecode for type safety and that there are no stack over- or underflows. The verifier simulates the execution of a method, but instead of operating on values, it operates on types. It is a fixpoint iteration on the types in the registers and on the stack of a method. Once the verifier reaches a fixpoint and there were no constraints violated, the method has successfully been verified. Note that termination of the method is not guaranteed, the verification algorithm may terminate even if the Java code does not.

In our case, we want all types in the registers to be the types (including the Universe type) of the corresponding parameters or local variables once the fixpoint is reached. Since we require a LocalVariableTable to be present, we know the scope of each local variable and can therefore internally use different variables if registers are reused. A register is therefore accessed with two arguments, the register index and the bytecode location of the current operation. The algorithm to annotate a single method works as follows:

A worklist is kept with all the instructions that need to be looked at. Initially, only the first instruction of the given method is in the worklist. The types (including Universe types) of the method arguments are stored in the corresponding registers. The special reference to the **this** object in register '0' has a special Universe type '`this_peer`'. A simple **peer** Universe type would not be correct, because the type combinator does not need to be applied for the **this** reference. As soon as the **this** reference is stored into another register, though, the Universe type is changed to a common **peer** type. If no Universe type was set for a parameter yet, it is left in the *unset* state (this may happen if a method was not called at all during the test case). The operand stack of the method is empty. The abstract interpreter executes the following loop:

1. Select a virtual machine instruction that is in the worklist. If the worklist is empty, the method body has successfully been annotated. All types stored in the registers are the

inferred types for the corresponding local variables. There is one type stored for each local variable using a certain register depending on the scope of the local variable (given in the LocalVariableTable). Otherwise, select an instruction and remove it from the worklist.

2. Model the effect of the selected instruction on the operand stack and registers by doing the following [1] (make sure to apply the type combinator where it is necessary):

   - If the instruction modifies a register, record that the register now contains a new type. The new type is the smallest common supertype of the type stored in the register and the assigned type, unless the register was in uninitialized state (see definition of scs* in Section 5.4.2).
   - If the instruction uses values from the operand stack, ensure that the top values on the stack are of an appropriate type. Otherwise, insert a cast to the correct type (Check if the user set a default).
   - If the instruction expects a certain type in a register, assume that the specified register contains a value of the appropriate type (since we assume that the bytecode is correct).
   - If the instruction pushes values onto the operand stack, add the indicated types to the top of the modeled operand stack.

3. Determine the instructions that can follow the current instruction. Successor instructions can be one of the following:

   - The next instruction, if the current instruction is not an unconditional control transfer instruction (for instance **goto**, **return**, or athrow).
   - The target(s) of a conditional or unconditional branch or switch.
   - Any exception handlers for this instruction.

4. Merge the state of the operand stack and registers at the end of the execution of the current instruction into each of the successor instructions. In the special case of control transfer to an exception handler, the operand stack is set to contain a single object of the exception type indicated by the exception handler information.

   - If this is the first time the successor instruction has been visited, record that the operand stack and register values calculated in steps 2 and 3 are the state of the operand stack and registers prior to executing the successor instruction. Add the successor instruction to the worklist.
   - If the successor instruction has been seen before, merge the operand stack and register values calculated in steps 2 and 3 into the values already there. Add the successor instruction to the worklist if there is any modification to the values.

5. Continue at step 1.

In order to find the annotation of parameters that had an unknown Universe type prior to the method body inference, we thought of the following simplification: Un-annotated method parameters are set to **peer** if there is a write access anywhere in the body of the method and **readonly** if there is not. This makes sense for all methods that are not private, because for non-private methods a **rep** annotation of a method parameter makes no sense. For private methods, a **rep** annotation is used when there is a write operation. The default for new-statements and for fields is **rep**. For static method calls, we try to figure out the annotation of the call by looking at the passed arguments to the method. If there is a **peer** annotation in the method signature, the annotation of the call equals the annotation of the passed argument.

[1] A detailed list of all instructions and their effects on the registers and stack is provided in Appendix A

Example

Next, we demonstrate a small example of the method body inference algorithm for the method in Figure 5.2. The sample code is not of practical use, because it will result in an infinite loop, but the abstract interpretation algorithm can still be demonstrated.

```
class A{
  public void test(rep peer Object [][] param_arr){
    rep peer Object[] local_arr = param_arr[2];
    while (true){
      rep Object local_obj = local_arr [2];
    }
  }
}
```

```
0:  aload_1
1:  iconst_2
2:  aaload
3:  astore_2
4:  aload_2
5:  iconst_2
6:  aaload
7:  astore_3
8:  goto 4
```

Figure 5.2: Source- and bytecode of the method body to be annotated. The two local variables local_arr and local_obj are to be annotated. The parameter param_arr has been assigned a **rep peer** annotation by the previous steps of the Inference tool. Note that the code does not terminate, but the abstract interpretation algorithm does.

The annotation for the parameter param_arr has been set to **rep peer** by the previous steps of the Inference tool. The initialization of the stack and the registers is therefore the following: register '0' is initialized with type this_peer A (the **this** object), register '1' with **rep peer** Object [][] (parameter param_arr). Registers '2' and '3' are initialized with the uninitialized type, denoted by '$\top$'. Table 5.3 shows the execution of the described algorithm. In this case, each instruction has exactly one successor. Except for the goto instruction (8), the successor is the next instruction in the bytecode. This means that one instruction will be in the worklist at a time. The outgoing frame situation of each bytecode instruction of the example in Figure 5.2 is displayed in the table.

When the algorithm reaches the **goto** instruction on line 8 whose successor has been visited before, the outgoing frame situation has to be merged with the present incoming frame situation of instruction 4 (i.e. outgoing frame of instruction 3). Other than register '3', which contained the uninitialized type before, all registers are the same. We have to apply our special smallest common supertype function (described in Section 5.4.2) to this register, which will yield **rep** Object.

The algorithm will then process instructions 5-8 again. When instruction 8 is processed this time, the incoming frame situation for instruction 4 will be the same as before and the instruction is not put into the worklist. The worklist is now empty, and the types of the local variables are stored in the corresponding registers (**rep peer** Object[] for local_arr , **rep** Object for local_obj).

5.4.2 Detailed Description of the Algorithm

Determination of Successor Instruction

The determination of successor instructions works exactly the same as for regular bytecode verifiers. For most instructions, the successor instruction is simply the next instruction in the bytecode. For branches and switches the successors are all possible targets. There is a pair of instructions that usually creates some problems with bytecode verifiers, the jump to subroutine instruction jsr and its return instruction ret. The target of the jsr is simply the target supplied as argument to the instruction. The ret instruction, however, retrieves the return address from a register, therefore the return address is not statically known. The Java Virtual Machine Specification (Section 4.9.6 [25]) states the following on ret instructions: "When executing the *ret* instruction, which implements a return from a subroutine, there must be only one possible subroutine from which the instruction can be returning. Two different subroutines cannot "merge" their execution to a single *ret* instruction." This means that we can assign to each ret instruction a set of corresponding

Instruction	Outgoing frame situation (Stack, Registers)
init:	([ ],[this_peer A, rep peer Object[][], ⊤, ⊤])
0: aload_1	([rep peer Object[][]], [this_peer A, rep peer Object[][], ⊤, ⊤])
1: iconst_2	([int, rep peer Object[][]], [this_peer A, rep peer Object[][], ⊤, ⊤])
2: aaload	([rep peer Object[]], [this_peer A, rep peer Object[][], ⊤, ⊤])
3: astore_2	([ ], [this_peer A, rep peer Object[][], rep peer Object[], ⊤])
4: aload_2	([rep peer Object[]], [this_peer A, rep peer Object[][], rep peer Object[], ⊤])
5: iconst_2	([int, rep peer Object[]], [this_peer A, rep peer Object[][], rep peer Object[], ⊤])
6: aaload	([rep Object], [this_peer A, rep peer Object[][], rep peer Object[], ⊤])
7: astore_3	([ ], [this_peer A, rep peer Object[][], rep peer Object[], rep Object])
8: goto 4	([ ], [this_peer A, rep peer Object[][], rep peer Object[], rep Object])
goto 4 => merge with outgoing frame situation of instruction 3; instruction 4 is put in the worklist.	
4:	([rep peer Object[]], [this_peer A, rep peer Object[][], rep peer Object[], rep Object])
5:	([int, rep peer Object[]], [this_peer A, rep peer Object[][], rep peer Object[], rep Object])
6:	([rep Object], [this_peer A, rep peer Object[][], rep peer Object[], rep Object])
7:	([ ], [this_peer A, rep peer Object[][], rep peer Object[], rep Object])
8:	([ ], [this_peer A, rep peer Object[][], rep peer Object[], rep Object])
goto 4 => merge with outgoing frame situation of instr. 3. The result is the same as before, so successor 4 is not added to the worklist again. The worklist is now empty and the inferred types are stored in the corresponding registers.	

Table 5.3: Abstract interpretation of the operand stack and registers. The outgoing frame situation of each instruction is displayed. The algorithm executes from top to bottom, because every instruction has exactly one successor. The successor of the **goto** instruction (inst. 8) is instruction 4. Since this instruction has been visited before, the outgoing frame situation of instruction 8 is merged with the other predecessor of instruction 4. The result is compared to the frame situation that was already there. If it is the same, the instruction will not be put into the worklist (which is the case when the algorithm reaches this instruction for the second time).

jsr instructions, which all have the same target. The instructions that follow these jsr instructions are successor instructions of the ret instruction.

State Merge

There is a difference between our algorithm and the algorithm for bytecode verifiers when merging the state of the operand stack and registers. Bytecode verifiers will always assign the smallest common supertype (according to the type hierarchy shown in Figure 5.3) of two types that are stored in the same register or at the same place on the operand stack. This means that whenever one of the two types that are merged is in the uninitialized state, the result of the merge is the uninitialized state. This is very impractical for our purpose, because we want the inferred type of a given local variable to be stored in the corresponding register upon termination of our algorithm. We do not care about uninitialized registers, because we assume that the given bytecode is correct. Therefore, the result of a merge with an uninitialized register and a type τ is always τ. Additionally, we have to include the Universe type when determining the smallest common supertype.

For bytecode verifiers, the jsr and ret instructions pose some problem at this stage again. The problem is that upon entering the subroutine, all output states of corresponding jsr instructions are merged. This can lead to loss of precision in the register types inferred, as the example in Listing 5.4 taken from Leroy[18] shows.

At the target of the two jsr instructions at 0 and 52, their output states are merged into a single state. This means that if the type of register 0 was "uninitialized" when the subroutine was called from 0 and "int" when called from 52, its state will be merged to "uninitialized" by the bytecode verifier. After returning from the subroutine, the "uninitialized" state of register 0 is preserved and the code following the jsr instruction at 52 will be rejected. For this purpose, the

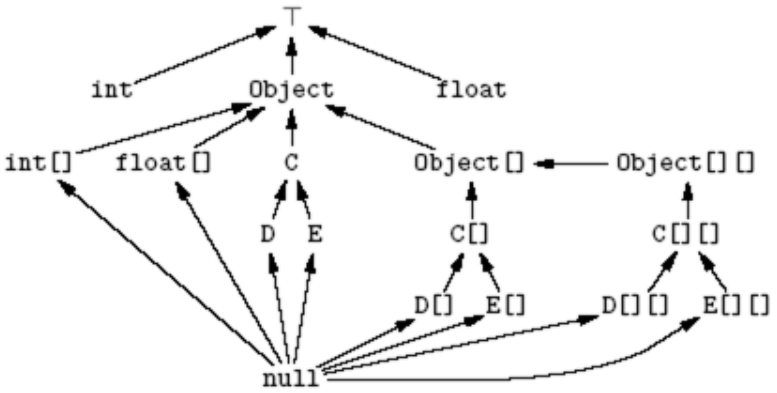

Figure 5.3: Some type expressions with their subtyping relationship. C, D, E are user-defined classes, with D and E extending C. "⊤" represents the "uninitialized" state or type. This figure is taken from Leroy[18].

Sun's bytecode verifier keeps track of all registers that are used within a subroutine and only those registers are merged, the rest stays unaffected. However, for our purpose this is not necessary, because we do not care about uninitialized registers and our special merge function scs∗() will never revert a register back to the uninitialized state.

Listing 5.4: Subroutine Example

```
                    // register 0 uninitialized here
 0: jsr 100         // call subroutine at 100
 3: ...

50: iconst_0
51: istore_0        // register 0 has type "int" here
52: jsr 100         // call subroutine at 100
55: iload_0         //load integer from register 0
56: ireturn         //and return to caller
        ....
                    //subroutine at 100
100: astore_1       //store return address in register 1
101: ...            //execute some code that does not use register 0
110: ret 1          //return to caller
```

Effects of Bytecode Instructions

Each instruction of a method has a certain effect on the registers and the stack of this method, this is modeled by a stack and register state transition rule. A complete list of all transition rules for every bytecode instruction is found in Appendix A. At this point we will only discuss some interesting instructions that have implications on the Universe types of the types on the stack or in the registers.

In order to explain the transition rules, the following helper functions are used:

$$
\begin{aligned}
om(\tau) &= Ownership\ modifier\ of\ type\ \tau. \\
jt(\tau) &= Java\ type\ of\ type\ \tau. \\
tc(\tau_1, \tau_2) &= om(\tau_1) * om(\tau_2)\ jt(\tau_2) \\
comp(\alpha) &= Component\ type\ of\ array\ \alpha. \\
\tau <: \tau\prime &= \tau\ is\ a\ subtype\ of\ \tau\prime.\ (Ownership\ modifier\ included) \\
scs * (\tau_1, \tau_2) &= \begin{cases} \tau_2\ if\ \tau_1 = uninitialized \\ \tau_1\ if\ \tau_2 = uninitialized \\ smallest\_common\_supertype(\tau_1, \tau_2)\ if\ \tau_i \neq uninitialized \end{cases}
\end{aligned}
$$

The selected rules in Figure 5.4 are interesting because they all differ in some way from the rules that a bytecode verifier would normally use. The astore instruction stores an object reference that is on the stack into a given register. This means that the reference needs to be on the stack before the instruction (denoted by $\tau.S$) and the smallest common supertype of the value already in the given register and the type on the stack, τ, is stored into register n after the instruction (denoted by $R\{n \leftarrow scs * (\tau, R(n))\}$). A bytecode verifier would simply store the new type in the register.

The aastore instruction requires three arguments to be on the stack. The array type α, the index into the array (as **int**) and the component type. For the component type, the Universe type combinator needs to be applied. The actual type that needs to be on the stack is given by the Universe type combination of the first and the second Ownership modifier of the array.

The invokevirtual instruction is very interesting, because for each type in the signature of the invoked method, the Universe type combinator with the type of the target object needs to be applied. This simply means that the types on the stack need to be compliant with the signature of the invoked method relatively interpreted to the target object. A small example: suppose the signature of a method contains only **peer** types. If you invoke this method on a **rep** object, all parameters need to be **rep**, too. This is because the arguments need to be **peer** relative to the target object. Furthermore, if a method uses **rep** anywhere in its signature, it may only be invoked on the **this** object. This is specified by the last condition of the rule. Of course, non-pure methods may only be invoked on types that are not **readonly**.

The last rule in Figure 5.4 for the instruction checkcast illustrates the effect of an existing cast. These are casts that were not added by the Inference tool, but where already present in the given program. Statically, the cast is always thought to succeed; the Ownership modifier remains unaffected. If the cast really succeeds can only be checked at runtime.

astore n : $(\tau.S, R) \rightarrow (S, R\{n \leftarrow scs * (\tau, R(n))\})$
aastore : $(tc(\alpha, \tau).\mathbf{int}.\alpha.S, R) \rightarrow (S, R),\ comp(\alpha) = \tau \wedge own(\alpha) \neq readonly$
invokevirtual C.m.σ : $(tc(\tau\prime, \tau_n\prime)...tc(\tau\prime, \tau_1\prime).\tau\prime.S, R) \rightarrow (tc(\tau\prime, \tau).S, R)$
$if\ \sigma = \tau(\tau_1, ..., \tau_n),\ \tau\prime <: C,\ \tau_i\prime <: \tau_i\ for\ i = 1...n,\ om(\tau\prime) \neq readonly,$
$if\ \exists\ \tau_i\ with\ own(\tau_i) = rep\ then\ om(\tau\prime) = peer\_this$
putfield C.f.τ : $(tc(\tau_2, \tau_1).\tau_2.S, R) \rightarrow (S, R)\ if\ \tau_1 <: \tau, \tau_2 <: C, om(\tau_2) \neq readonly$
checkcast τ : $(\tau\prime.S, R) \rightarrow (\tau.S, R),\ \tau <: \tau\prime,\ om(\tau) = om(\tau\prime)$

Figure 5.4: Selected stack and register state transition rules. They illustrate the effect of a bytecode instruction on the stack and register pair (S, R). The types on the stack are separated by '.'.

5.5 Implementation

Since the abstract interpretation approach for the annotation of method bodies is very similar to the abstract interpretation performed by bytecode verifiers, it was intriguing to use parts of an

open source verifier. Open source implementations of the Java Virtual Machine should provide a bytecode verifier, so we have taken a look at them first. Namely, we evaluated kaffe[26] and sablevm[12]. The sablevm did not seem to include a bytecode verifier at all. Kaffe includes a verifier implementation even though they state on their homepage that it does not. However, the code is written in C and very much tied into the rest of the virtual machine code, so it is not very useful for our purpose.

The bytecode engineering library BCEL includes a bytecode verifier called JustIce. JustIce is entirely written in Java and uses the bytecode engineering facilities provided by BCEL. JustIce turned out to be a good starting point for our abstract interpreter as we could use a lot of its code. JustIce uses the visitor pattern to go through the code of the methods to be verified, so we can specify our own visitors that can be used to annotate method bodies. Furthermore, it should be possible to improve or modify our abstract interpretation implementation later on such that a Universe type bytecode verifier can be produced. The highest benefit from using BCEL and JustIce was that we did not have to care about the successor problem of jsr / ret pairs and the program flow in general. We could simply rely on the verifier facilities. As we have shown in Section 4.4, we use the bytecode engineering tool javassist to implement the instrumentation of loaded classes. Since these two libraries are not loaded at the same time (javassist is used in the Tracing agent, BCEL in the Type inferer), we do not introduce any runtime overhead by using both libraries.

The separate abstract interpretation step is tied into the existing system as an additional visitor (class AbstractInterpretationVisitor). This way, the method body annotation is optional and can be deactivated by removing its entry in the configuration file. This would make sense when library code is examined which does not provide any source code that could be annotated.

At first, we wanted to run a normal bytecode verification prior to the method body inference to be sure that the given bytecode is correct. Unfortunately, the JustIce version we used (as part of BCEL 5.1) had some problems with interfaces and abstract classes, so we could not rely on the verifier result (the verifier rejected correct bytecode).

5.5.1 Java type representation

To represent Java types on the stack and in the registers, the BCEL uses the following classes (see Figure 5.5): BasicType, ReturnaddressType, and ReferenceType. Furthermore, there are two kinds of ReferenceTypes: ObjectType, which stands for all kinds of class types and ArrayType, which stands for all array types. Of course, these classes consider no Universe modifiers, so we had to add new classes that do. Since only reference types may have Universe modifiers, we added a UniverseReferenceType, a UniverseObjectType and UniverseArrayType. The most convenient way to add our classes storing Universe type information to the existing class hierarchy would have been to make UniverseObjectType a subclass of ObjectType and UniverseArrayType a subclass of ArrayType. This way, we would have achieved the best compatibility with the existing JustIce code, as it did not have to deal with any Universe type information and our additional classes could use this information. However, those two types are declared **final** so we could not do it this way. Instead, we had to make UniverseReferenceType a subclass of ReferenceType and the array and object types inherit from this one. This made the rest of the implementation significantly more difficult, as **instanceof** checks of either ArrayType or ObjectType within the JustIce code did not work anymore. We therefore could not inherit from the code visitors, but had to implement our own doing a lot of copy and paste of existing JustIce code.

5.5.2 Verification visitors

The JustIce bytecode verifier implements the verification algorithm described in the JVM Spec, which we customized for our purpose (see Section 5.4.1). We had to alter the method frame representation to be able to deal with our new UniverseReferenceTypes and had to make sure that each local variable stored in a register can be accessed separately. Constraints of a bytecode instruction are checked by the InstConstraintVisitor . This visitor checks all constraints of an instruction, e.g.

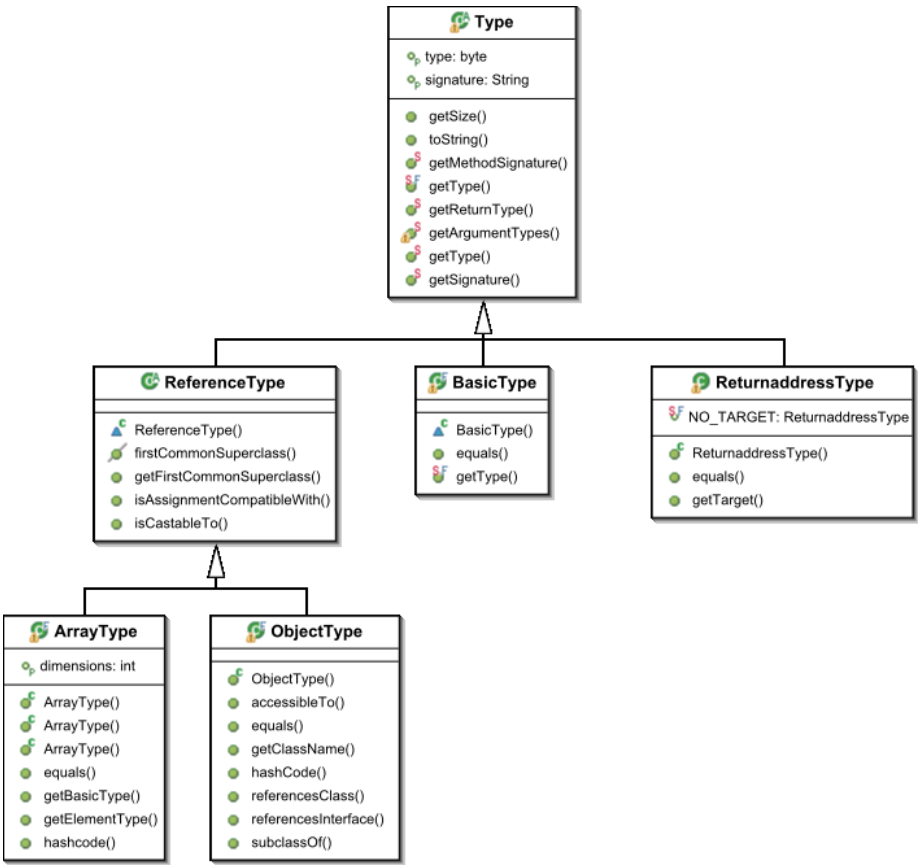

Figure 5.5: BCEL classes to simulate types on the stack and in the registers.

if there are enough objects on the stack and so forth. Our implementation only checks Universe type constraints, e.g. a non-pure method invocation needs to have a target on the stack that is not **readonly**. Once the constraint visitor ensured that the given constraints are met, the effect of the instruction is modeled, i.e. the instruction is virtually executed using the ExecutionVisitor. Here, we simply copied all methods that visited an instruction that was not using any Universe types and reimplemented the ones using Universe types.

5.5.3 Indexing within Method Bodies

As specified in the annotations.xsd schema[2], the output of the Type inferer needs to index static calls, new-statements, local variables, and casts within a method body. For each target class of a static call and for each new-statement one index is maintained, e.g. all new-statements are assigned an increasing index as they appear in the source code of a method body. Local variable declarations are indexed in the order they appear in the source code.

The indexing of casts is somewhat less trivial. In general, there are three types of cast targets: assignments, method calls, and array initializers. Each target type is indexed throughout the method body, this number is called the *index*. Within a cast target, there may be a number of objects that can be casted (we call this the *position*): the target of an assignment or a method call is at position -1, the n parameters of a method call are numbered from 0 through (n-1). For an assignment, the expression that is assigned to the target is at position 0. The example in Listing 5.5 shows a method body with 4 assignments and one method call.

[2]see Appendix D line 387ff

Listing 5.5: Example of cast numbering within a method. All assignments and method calls are numbered from top to bottom of the method (this number is called the *index*). Within a method call or assignment, the position of the cast must be specified (this number is called the *position*). E.g. the cast of the target of the method call on line 5 would get $index = 0$, $position = -1$.

```
1 public void foo(){
2         int x= 5;         //type:assignment, index=0
3         readonly Object y= new Object();         //type:assignment, index=1
4         XY obj= (readonly XY) obj_2;    //type:assignment, index=2, position= 0
5         ((peer XY) obj).bar(y, x);         //type:method call, index=0, position= -1
6         ((peer XY) obj). field = new Object();    //type:assignment, index=3, position= -1
7 }
```

Once a method's bytecode is parsed, we go through the code array and assign the indexes of static calls, cast targets, and new-statements. We do this by using the information in the LinenumberTable, which can be used to look up the source code line number of each bytecode instruction. However, if two statements are on the same line, we do not know in which order they appear in the source code. This means that the Type inferer might mix up statements on the same line (as pointed out by Figure 5.1 in Section 5.4). This can be resolved by separating different statements on different lines.

Furthermore, indexes for constructors may cause additional problems, because their instance initializers are inlined into all constructors during compilation (see Listing 5.6). In the example, the object construction on line 6 will get the index 1, the array construction on line 2 will get the index 0. This behavior does not comply with the specification in the annotations.xsd schema, instance initializer code should be numbered independently. Handling this problem remains future work.

Listing 5.6: Instance initializers, such as the statement on line 2 are inlined into the constructors during compilation. In the bytecode, it cannot be seen if the construction of the array is done in the constructor or as part of the instance initializer.

```
1         class X1{
2                 Object[]  array= new Object[4];
3                 Object obj;
4                  ...
5                 public X1(){
6                         obj= new Object();
7                 }
8         }
```

In our implementation using the BCEL library, we did not find a way to index local variables correctly. BCEL only provides information about the register slot in which a certain variable is stored, not the information at which position in the source code a local variable is declared. All entries of a local variable table are stored in an array. It seems that most of the time, the order in the array and the order in the source code is the same. However, there are cases where this order is not the same and the tool will produce wrong local variable indexes. Since we also output the name of the local variable, though, the user can still correct the result by hand or the annotation tool could use the variable name instead of the index to insert the annotations. Using the variable names for the annotation tool requires that no name is used multiple times within a method body.

Chapter 6

Joining multiple Test runs

Code coverage is crucial for the quality of the annotations found by the Inference tool. Even though the abstract interpretation for the method body inference can deal with code that has not been visited to some extent, the abstract interpretation might get stuck and no valid annotation can be found in some cases. In this chapter, we show how multiple test cases can be joined, resulting in better code coverage and hence less unannotated variables prior to the abstract interpretation phase. In Section 6.1 we show how global information, valid for all test cases, and local information, valid for just one test case, is separated. The implications of this separations on the visitors is presented in Section 6.2. The chapter is concluded with a presentation of a test case that shows the benefits of the possibility to join multiple test runs in Section 6.3.

6.1 Information separation

Until now, we have only dealt with one program trace. Meaning only one EOG needed to be built up and all operations were performed on this graph. When multiple traces need to be joined, it is not possible to build up a single EOG. This will not work, because object IDs assigned by the Tracing agent are only unique for one run. The same ID can be used for different objects in two runs which will cause errors when building up the EOG. In Version 1, all information about variables, mappings from trace file IDs to objects in the EOG, etc. was handled by the same class Graph. If we want to have multiple EOGs, we need to separate global and local information. Here is a list of global and local information to be separated:

1. Global information:
    - Pure methods, default annotations
    - Variables to be annotated
    - Variable → Reference mapping (needed in harmonization phase)
2. Local information:
    - References and objects of the EOG
    - Field/Method ID → java.lang.reflect.Field/java.lang.reflect.Method mapping
    - Object ID → object mapping

Since we now have multiple graphs, it makes sense to use the existing Graph class as representation of a single EOG and add a new layer above the existing, dealing with all global information. The UML diagram of the new class ProgramTraces and the changes to the existing Graph class can be seen in Figure 6.1. We have changed the name of the observer interface, such that it is clear that the observers observe a ProgramTraces object, not a Graph object. We have added the method graphAdded() to the interface, which will be called when a new graph is created. Other than that,

we have not changed the interface to ensure that only minor changes need to be applied to the existing tools such as Marco Meyer's EOG visualization tool[20].

6.2 Visitors

In Version 1 of the tool, all visitors were operating on the single Graph object, since it contained all information about the variables and the EOG. Now that we have separated information locally relevant for each EOG and information globally relevant for the variables to be annotated, we can classify the existing visitors into three categories:

1. Visitors using only local information about an EOG:
   - DominatorVisitor
   - StoreDominatorLevelVisitor
   - ResolveConflictsVisitor
2. Visitors using only global information about annotation variables:
   - AbstractInterpretationVisitor
   - OutputVisitor
3. Visitors using both global and local information:
   - BuildUpVisitor
   - HarmonizationVisitor

Even though we have pointed out that there are different kinds of visitors, now that we have separated some data, we did not change the visitor interface (except a name change from GraphVisitor to TraceVisitor). All visitors visit objects of the same class ProgramTraces. This still makes sense, because only ProgramTraces objects keep track of how many EOGs there are in the system.

Here is how the different steps of the algorithms are executed now that multiple trace files are supported:

The build-up phase of the algorithm consists of two steps: first, the global information such as default annotations and pure methods is parsed and registered with the ProgramTraces object. Then, one EOG is locally built up for each trace file that was specified as input.

Once all EOGs are built up, the three local visitors (Dominator−, StoreDominatorLevel−, and ResolveConflictsVisitor) resolve local conflicts until a valid object hierarchy is established in each graph.

The HarmonizationVisitor then needs to harmonize global conflicts. Global conflicts might occur if one EOG suggests a **peer** annotation of a variable while another suggests a **rep** annotation. This conflict, for example, needs to be resolved by making objects connected by the given variable references in all graphs **peer** to each other. In order to be able to identify an object's or reference's graph, a reference to the graph is stored in each object and reference. This increases the total memory usage slightly, but is necessary to resolve global conflicts. This reference can also be used by observers to locate the graph of an object.

The AbstractInterpretationVisitor and OutputVisitor only use the information stored in the ProgramTraces object (i.e. inferred and default annotations of the variables). If memory usage was an issue, all references to the EOGs and their objects could be nullified such that their memory could be freed. We have not chosen to do this, because the abstract interpretation and output steps do not need excessive amounts of memory themselves.

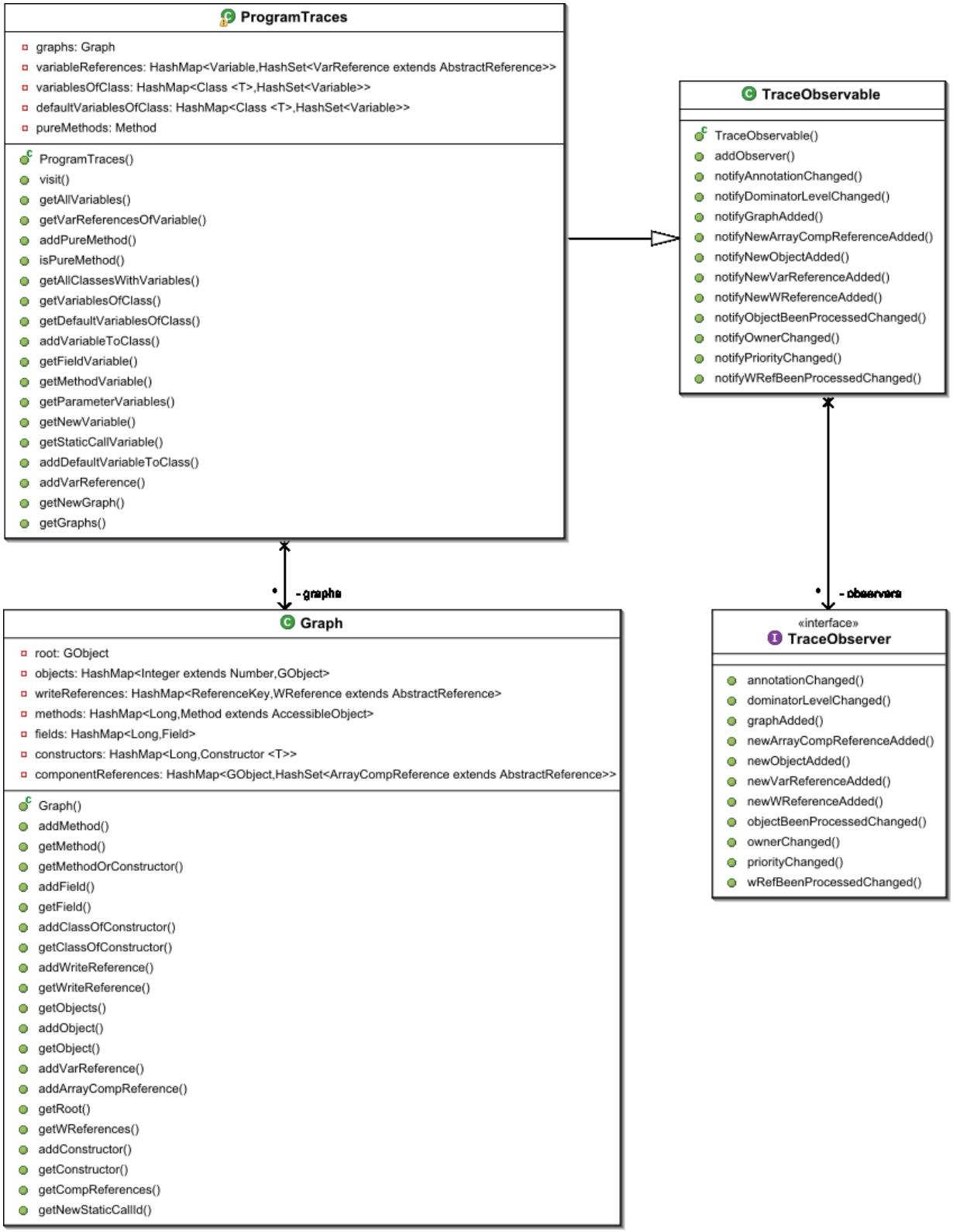

Figure 6.1: The new class ProgramTraces deals with global information that is shared by all graphs, e.g. variable information, pure method information, etc., while class Graph now holds local information that is only relevant for a particular EOG. The names of the observer classes has been changed to indicate that not a single EOG (i.e. class Graph) is observed anymore, but the class ProgramTraces.

6.3 Merge Example

To demonstrate the benefit of merging several test runs into a single Inference run, we tested the example in Listing 6.1. We ran the example two times, once the argument for the method test() was "true", once it was "false".

Listing 6.1: Class to test the merging of several test runs. We ran the method test() once with the argument "true", once with the argument "false", producing two trace files.

```
1  public class Test {
2          public Test test_field ;
3
4          public Test(){
5          }
6
7          public void test(boolean bool){
8                  Test local_test ;
9                  if (bool){
10                         test_field = new Test();
11                 }else{
12                         test_field = new Test();
13                         test_field .makePeer(this);
14                 }
15                 local_test = test_field ;
16         }
17
18         //non-pure method call, i.e. write operation
19         public void doWrite(){
20         }
21
22         //makes "other" peer to this
23         public void makePeer(Test other){
24                 other.doWrite();
25         }
26
27         public static void main(String[] args) {
28                 Test test= new Test();
29                 test.test(true);          //second run with argument "false"
30         }
31 }
```

If we use just the first run as input, we get a **rep** annotation for the field test_field , the local variable local_test , and both new-statements within method test(). The new-statement in the "else" branch defaults to **rep**, because it is never called during the test run. The parameter other of method makePeer defaults to **peer**, because it is a public method. This makes sense, because the method performs a write operation on the parameter. However, the method body annotation of test() does not terminate and aborts with an error. When the method invocation on line 13 is processed, the parameter of the method must be interpreted relative to the target object test_field (which is **rep**), so the supplied argument must be **rep** as well. Since the actual argument (this) currently on top of the stack is not **rep**, the algorithm checks if a cast is possible. There is no way to cast a **peer** object to a **rep** object, so the abstract interpretation fails. This example cannot be made working by setting default annotations, because a default annotation may only be set for the new-statement in the "else" branch. Any other default would be overridden by the runtime annotation, since defaults are only regarded in the method body annotation step. All other variables (e.g. the field test_field) already have been annotated at that time.

If we use just the second run as input, we get a **peer** annotation for the field test_field and for the new-statement in the "else" branch. The new-statement in the "if" branch defaults to **rep**, which causes the abstract interpretation to fail again (after it tries to assign the newly created **rep** object to the **peer** field test_field). However, this time we can make it work by supplying a default **peer** annotation for the new-statement in the "if" branch. The result when we do this is the same as the one presented in the next paragraph using both runs as input.

If we use both runs as input we get a valid, compilable solution without further interaction. Both new-statements, the local variable, as well as the field test_field are annotated with **peer**. The example shows that bad code coverage may lead to wrong results and the abstract interpretation may get stuck finding no valid solution. In these cases either code coverage must be improved, or good defaults must be set. As we have seen in this chapter, code coverage may be improved significantly by using several test cases as input to the Type inferer.

Chapter 7

Results and future work

In this chapter we discuss the results of this thesis. Some examples that demonstrate the functionality of the tool are presented in Section 7.1. In Section 7.2 we take a look at some related work. Possible future work that could improve the existing tool is presented in Section 7.3. Finally, we will draw the conclusion in 7.4.

7.1 Program Examples

We have tested the Inference tool with some simple examples focusing on the new features implemented. In the end of this section, we discuss a larger example that uses interfaces, abstract classes, and inheritance.

7.1.1 Producer Consumer

The Producer/Consumer example (see Listings 7.1, 7.2) is taken from the article "Universes: Lightweight ownership for JML" [9]. The producer writes products into the shared buffer, whereas the consumer reads from this buffer. The producer, the consumer, and the products are in the same context, because they are all created by the static main() method (see figure 7.1). The producer and the consumer store a reference to each other.

After running a test case producing 100 products and directly consuming them, we got the same result as [9]: The producer and the consumer are **peer** to each other, while the buffer array is in the **rep** context of the producer. Therefore, the consumer only gets a **readonly readonly** reference to the buffer. Then we added a write access from the consumer to an object referenced by the buffer array (on line 16 in Listing 7.2). This write access did not change the annotation of the array, because we cannot add a write reference to the array (see Section 4.2). However, the write access is allowed, because the object referenced by the array is in the same context as the consumer. The method body annotation algorithm added a cast to **peer** as expected, so the result is compilable.

Listing 7.1: Producer.

```
1  public class Producer {
2
3          public rep readonly Product[] buf;
4          public int n;
5          public peer Consumer con;
6
7          public Producer() {
8                  buf = new rep readonly Product[10];
9          }
10
```

```
11          public void produce(peer Product p) {
12                  buf[n] = p;
13                  n = (n+1) % buf.length;
14          }
15
16          public static void main(readonly readonly String args){
17                  peer Producer producer= new peer Producer();
18                  peer Consumer consumer= new peer Consumer();
19                  for (int i=0; i<100; i++){
20                          peer Product p= new peer Product();
21                          producer.produce(p);
22                          consumer.consume();
23                  }
24          }
25  }
```

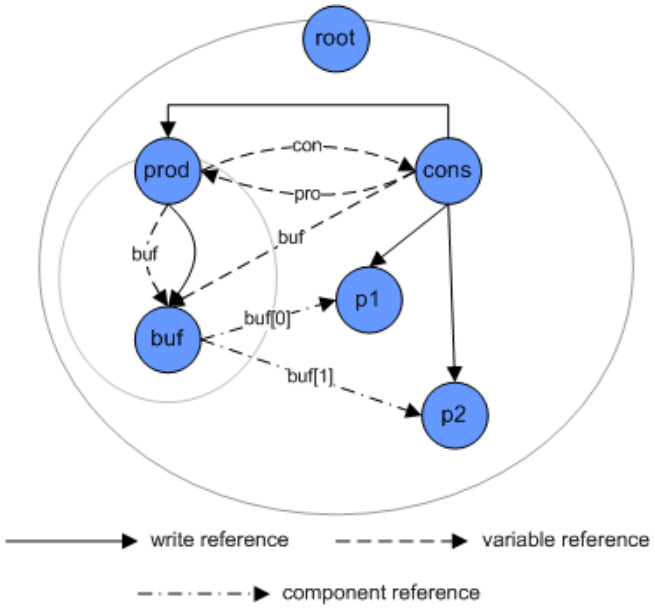

Figure 7.1: The Producer prod, the Consumer cons and the Products p1 and p2 are in the same context. The buffer array buf is in the context owned by the Producer. Write operations on the products by the Consumer using the buffer array variable is legal even though the array's Universe type is **readonly readonly**.

Listing 7.2: Consumer.

```
1   public class Consumer {
2
3           public readonly readonly Product[] buf;
4           public int n;
5           public peer Producer pro;
6
7           public Consumer(peer Producer p) {
8                   buf = p.buf;
9                   pro = p;
10                  n = buf.length-1;
11                  p.con = this;
12          }
13
```

```
14          public readonly Product consume() {
15                  n = (n+1) % buf.length;
16                  //modification: ((peer Product) buf[n]). field = 10;
17                  return (readonly Product) buf[n];
18          }
19  }
```

7.1.2 Array Example

The Producer/Consumer example already included the annotation of an array. In this ArrayTest example (Listing 7.3), we have two fields of array type and one local variable of array type. We have also added a method that will never be called during the test run, to see how our tool handles this. Version 1 would not have generated any annotations for that method.

Listing 7.3: Example to test some array annotations. Two fields of array type rep_peer_array and peer_peer_array are created and filled. Peer_peer_array must be annotated with **peer peer** because an object that is peer to this (peer_other) writes to the array as well.

```
1   public class ArrayTest {
2
3           private rep peer ArrayTest [][]  rep_peer_array ;
4           private peer peer ArrayTest []  peer_peer_array ;
5
6           private peer ArrayTest  peer_other ;
7
8           public void test (){
9                   readonly peer ArrayTest []  local_array ;
10                  rep_peer_array = new rep peer ArrayTest [3][4];
11                  for (int i=0; i<3; i++){
12                          for (int j=0; j<4; j++){
13                                  rep_peer_array [i][j]= new rep ArrayTest();
14                          }
15                  }
16                  local_array = rep_peer_array [1];
17                  peer_peer_array = new peer peer ArrayTest[4];
18                  for (int i= 0; i<4; i++){
19                          peer_peer_array [i]= new peer ArrayTest();
20                  }
21                  peer_other= new peer ArrayTest();
22                  peer_other .makePeer(this);
23                  peer_other .writeToArrayComponents(peer_peer_array);
24                  peer_other .writeToArray( peer_peer_array );
25                  if (peer_other == null){
26                          peer peer ArrayTest []  new_array= not_called( peer_peer_array );
27                          peer_peer_array [0]= new_array [0];
28                  }
29                  local_array = peer_peer_array ;
30          }
31
32          //writes to all array components
33          public void writeToArrayComponents(peer peer ArrayTest[] arr){
34                  for (int i=0; i<arr.length ; i++){
35                          arr [i ]. doWrite();
36                  }
37          }
38
39          //write operation on the array
40          public void writeToArray(peer peer ArrayTest []  arr ){
```

```
41                  arr[0]= arr [1];
42          }
43
44          //A method that is never called in the test case
45          public ArrayTest [] not_called (peer peer ArrayTest [] arr){
46                  arr[0]=new peer ArrayTest();
47                  return (peer peer ArrayTest []) arr;
48          }
49
50          //simulates a write operation
51          public void doWrite(){
52          }
53
54          //makes the calling object peer, if caller == other
55          public void makePeer(peer ArrayTest other){
56                  other.doWrite();
57          }
58
59          public static void main(readonly readonly String [] args){
60                  peer ArrayTest test= new peer ArrayTest();
61                  test . test ();
62          }
63  }
```

The tool produces the expected output. Field rep_peer_array is annotated with **rep peer** because the only object writing to the array is the one created by the static main() method. On the other hand, Field peer_peer_array is written to by two objects that are **peer** to each other. If two objects that are **peer** to each other write to the same object (the array in this case), all three object must be **peer**, otherwise at least one write reference would cross a context boundary (see Figure 7.2).

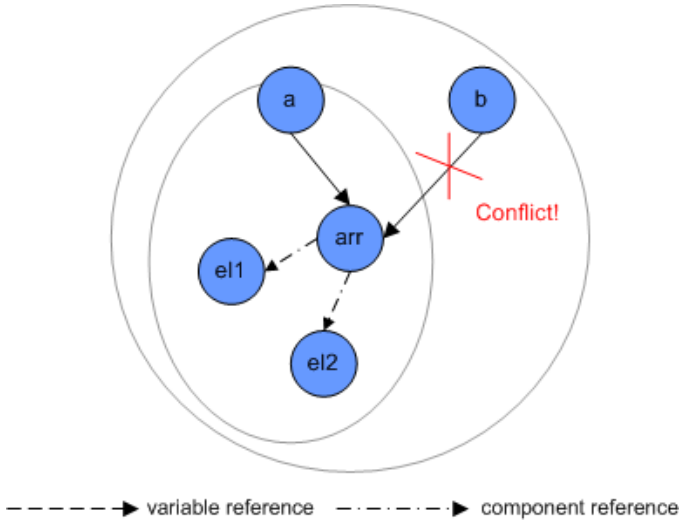

Figure 7.2: If two objects that are **peer** to each other (a and b) write to the same object arr, all objects must be in the same context. Otherwise there is at least one conflicting write reference (in this case from b to arr).

Local variable local_array of method test() is annotated with **readonly peer**, because both a **peer peer** array and a **rep peer** array are stored in that variable at some point during the program execution. Therefore, the smallest common super type of these two Universe types must be assigned to the local variable, which is **readonly peer**.

The method argument and the return type of method not_called() default to **peer peer**, which is correct in this case. However, a **peer** annotation must be set in the default annotations input file for the new-statement in this method (line 46), because the default used by the abstract interpretation algorithm (**rep**) would not work (the assignment to the **peer peer** array fails). If the write operation on the array on line 46 is omitted, the annotations of the parameter and return types become **readonly readonly**.

7.1.3 LinkedList

The linked list example presented in this section is taken from [19]. We used the same test case as Lyner in his work to be able to compare the results of the current version to the results of Version 1 of the Inference tool (see Listing 7.4). The implementation of the linked list is displayed in Listing 7.5. Note that there are two kinds of remove() methods: one takes an integer as parameter, the other takes a parameter of type Object. Method contains() of class ListItem and Object.equals() are marked as **pure** by the input file.

Our tool produces the same results as Version 1 for the signatures of the methods and for the fields of the classes. All list items belong to the context owned by the linked list. The stored objects are **readonly** to the items. However, Version 1 did not produce any output for the remove(**int**) method, because it has never been called in the test case. The current version infers a **readonly** annotation of the method return type during the method body inference, which is correct.

The method body annotation works well: all local variables and new-statements are inferred correctly. Object creations within the static main() method are annotated with **peer**. All new-statements in the implementation code use the same Ownership type as the variable into which the newly created object is stored.

Listing 7.4: Test case for the linked list example.

```
1  public class LinkedListUser {
2          public static void main(String[] args) {
3                  peer LinkedList list = new peer LinkedList();
4                  peer A a1 = new peer A();
5                  peer A a2 = new peer A();
6                  peer A a3 = new peer A();
7                  peer B b1 = new peer B();
8                  peer B b2 = new peer B();
9                  peer B b3 = new peer B();
10                 list.insert(a1);
11                 list.insert(a2);
12                 list.insert(a3);
13                 list.insert(b1);
14                 list.contains(a1);
15                 list.contains(b1);
16                 list.size();
17                 list.remove(a3);
18                 list.insert(b2);
19                 list.remove(a2);
20                 list.remove(b1);
21                 list.insert(b3);
22                 list.size();
23         }
24 }
```

Listing 7.5: Implementation of the linked list.

```
1  public class LinkedList {
2          rep ListItem head;
3          int size;
4
5          public LinkedList(){
6                  head = null;
7                  size = 0;
8          }
9
10         public int size(){
11                 return size;
12         }
13
14         public void insert(readonly Object o){
15                 if(head == null){
16                         head = new rep ListItem(o);
17                 }else{
18                         head.insert(o);
19                 }
20                 size++;
21         }
22
23         public boolean contains(readonly Object o){
24                 if(head != null){
25                         return head.contains(o);
26                 }else{
27                         return false;
28                 }
29         }
30
31         public readonly Object remove(int index){
32                 if(head == null || index < 0){
33                         return null;
34                 }else if(index == 0){
35                         readonly Object ret = head.stored;
36                         head = head.next;
37                         return ret;
38                 }else{
39                         return head.remove(index-1);
40                 }
41         }
42
43         public readonly Object remove(readonly Object o){
44                 if(head == null){
45                         return null;
46                 }else if(o.equals(head.stored)){
47                         readonly Object ret = head.stored;
48                         head = head.next;
49                         size--;
50                         return ret;
51                 }else{
52                         readonly Object ret = head.remove(o);
53                         if(ret != null){
54                                 size--;
55                         }
56                         return ret;
57                 }
```

```
58
59          }
60  }
61
62  public class ListItem {
63          readonly Object stored;
64          peer ListItem next;
65          public String name;
66
67          ListItem(Object toStore){
68                  stored = toStore;
69                  next = null;
70          }
71
72          public peer ListItem getNextItem(){
73                  return next;
74          }
75
76          public void insert(readonly Object toStore){
77                  if(next == null){
78                          next = new peer ListItem(toStore);
79                          next.name = "item";
80                  }else{
81                          next.insert(toStore);
82                  }
83          }
84
85          public readonly Object remove(readonly int index){
86                  if(next == null){
87                          return null;
88                  }else if(index == 0){
89                          readonly Object ret = next.stored;
90                          next = next.next;
91                          return ret;
92                  }else{
93                          return next.remove(index-1);
94                  }
95          }
96
97          public readonly Object remove(readonly Object o){
98                  if(next == null){
99                          return readonly null;
100                 }else if(o.equals(next.stored)){
101                         readonly Object ret = next.stored;
102                         next = next.getNextItem();
103                         return ret;
104                 }else{
105                         return next.remove(o);
106                 }
107         }
108
109         public pure boolean contains(readonly Object o){
110                 if(o.equals(stored)){
111                         return true;
112                 }else{
113                         if(next == null){
114                                 return false;
115                         }else{
```

```
116                             return next.contains(o);
117                         }
118                     }
119                 }
120 }
```

7.1.4 Tree

The tree example presented here is taken from [16]. It uses interfaces, abstract classes, and subclassing (see Figure 7.3). The source code of the example with all inferred annotations is found in Appendix B. A test class is displayed in Listing 7.6. As it can be seen in the code for the static main method, there are three ways to use this class: with one of the arguments "average", "sumsub", and "search int". Of course we used the possibility to join several trace files in this case, by using one trace file for each of the invocation possibilities.

We have had to separate the two new-statements on lines 15 and 16 so that the Inference tool does not mix them up. After the separation, the tool produced the correct result for class Test: all new-statements and local variables of static methods are annotated with **peer**.

The rest of the code was annotated as expected: The SortedTree class is the owner of all SortedTreeNodes. The two kinds of tree iterators (PostorderTreeIterator and InorderTreeIterator) only have a **readonly** reference to the elements of the tree (i.e. tree nodes). The PostorderTreeIterator is annotated correctly, even though there is never an instance of this class created in the given test case. However, method createPostorderIterator () of class SortedTree, which is never called either, creates the iterator in its **rep** context instead of the **peer** context like the createInorderIterator () method. The iterator is still usable from the outside if all its methods are **pure** . If this is not the case, a default can be set by the user in the default annotations input file.

We have experienced some problems with this example, though. Methods of interfaces were not annotated at all. This is the case, because there is never an instance of an interface; there are only instances of the implementing classes. This means that all events are triggered by implementing classes and the variables to be annotated are associated with those classes. The same thing happens for methods of abstract classes. Only fields of abstract classes are annotated, because they do trigger events.

The general problem is that events such as *method exit* creating a variable for the return value, store the method of the actual type of the object triggering the event in the variable. This means that if the method is inherited, the variable is not used to annotate a possible super implementation of the same method. To solve this problem, we should probably create an additional variable for each super implementation of the same method. These variables could then be annotated separately in the harmonization and annotation phase. There might be some harmonization necessary if the inferred types for the method signatures in different subclass implementations are not the same.

Listing 7.6: Test class for the Tree example.

```
1  public class Test {
2
3      public static void main(readonly readonly String[] argsv) {
4          peer Statistics s = null;
5          if (argsv.length < 1) {
6              return;
7          }
8          else if (argsv[0].equals("average")) {
9              s = new peer Average();
10         }
11         else if (argsv[0].equals("sumsub")) {
12             s = new peer SumAndSubtract();
13         }
14         else if (argsv[0].equals("search")) {
```

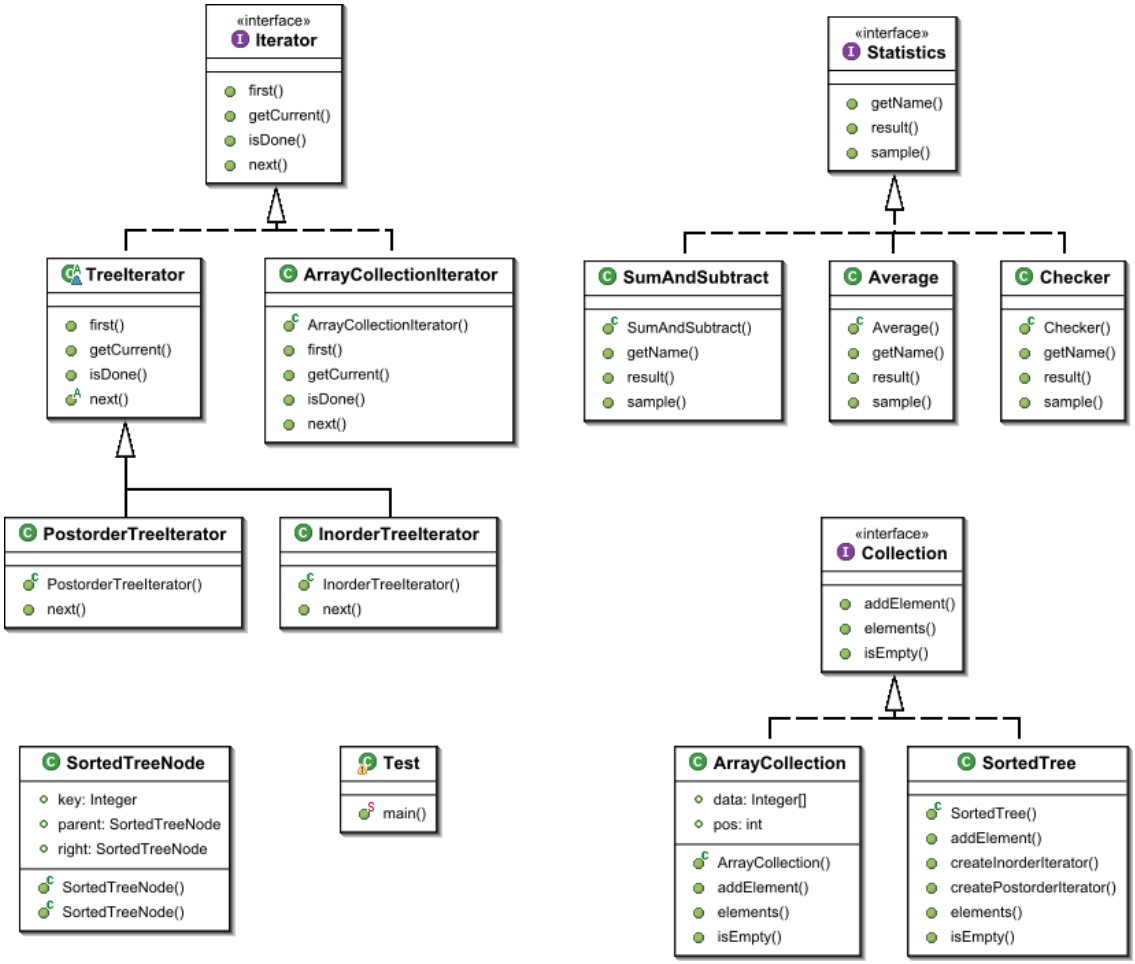

Figure 7.3: UML diagram of the Tree example taken from [16]. There are two collection implementations: an ArrayCollection and a SortedTree implementation. The elements of the SortedTree are stored in SortedTreeNodes while elements of the ArrayCollection are stored in a simple array.

```
15                      s = new peer Checker(
16                                      new Integer(Integer.parseInt(argsv[1])));
17              }
18              else {
19                      return;
20              }
21              Test.measure(10000, s);
22      }
23
24      private static void measure(int elements, peer Statistics s) {
25              int i;
26              long start, end;
27              start = System.currentTimeMillis();
28              peer ArrayCollection array = new peer ArrayCollection();
29              fill(array, elements);
30              apply(array, s);
31              peer SortedTree tree = new peer SortedTree();
32              fill(tree, elements);
33              apply(tree, s);
```

```
34                  end= System.currentTimeMillis();
35                  System.out. println ("Time:␣"+(end-start)+"ms");
36          }
37
38          private static void apply(peer Collection coll , peer Statistics s) {
39                  peer Iterator iter = coll .elements();
40                  while(! iter .isDone()) {
41                          s.sample( iter .getCurrent ());
42                          iter .next ();
43                  }
44          }
45
46          private static void fill (peer Collection coll , int elements) {
47                  peer Random random = new peer Random();
48                  random.setSeed(42);
49                  for(int i = 0; i<elements; i++) {
50                          coll .addElement(new Integer(random.nextInt() \% 100));
51                  }
52          }
53  }
```

7.2 Related work

The Master thesis by Nathalie Kellenberger "Static Runtime Inference" [16] and its follow-up "Static Universe Type Inference using a SAT-Solver" [22] by Matthias Niklaus deal with the inference of Universe types of a given program by analyzing the source code statically. The problem with the static inference is that the solution returned by the program may not be the optimal one, whereas the Runtime Inference tool will always return the best (i.e. most nested) solution. Since the static inference does not have to make the indirection via a trace file, its performance was much better than the one our runtime approach.

Marco Meyer has completed a semester project "Interaction with Ownership Graphs" [20]. The resulting program can be used together with the Type inferer to visualize the manipulations of the Extended Object Graph. If extended further, this program may allow the user to interact with the algorithm by resolving conflicts and other problems while the Inference tool is running.

Rayside, Mendel, and Jackson presented [8] a dynamic analysis for revealing object ownership. They use the owner as dominator property to find the ownership structure of all objects on the heap. However, they do not map this dynamic structure to a static structure such as source code. They implemented an interesting way to add write edges into the object graph that could be used to conduct a purity analysis while tracing the program flow. They also describe a caching mechanism that reduces the size of the trace file by about 50%, which could be implemented for our program as well.

Moelius and Souter [21] presented an algorithm for automatically identifying data sharing relationships in Java programs. Their implementation is based on escape-analysis.

Agarwal and Stoller [1] work on a combination of static and dynamic analysis to infer types for Parameterized Race Free Java [2]. Just like in our implementation, the dynamic analysis is used to infer the types for fields and method parameters, while the static analysis is used to infer local variable types. The Parameterized Race Free Java type system is focused mainly on multi-threaded programs.

Wren [27] has presented a theoretical foundation for a combination of runtime and static Ownership Inference. However, he does not present in implementation for his theoretical work.

7.3 Future work

Disk-I/O Since disk-I/O contributes to about 50% of the total execution time of the Inference tool, it may be worthwhile to improve the file exchange format and file generation mechanism. When we zipped the 115 MB trace file from the Linked List example mentioned in Section 5.3, the size was compressible down to 3 MB. This means that there is a lot of redundant data, which may be compressed right away. A caching mechanism for events may be implemented such that events that do not add any more information to the EOG are not written to the trace file. In our LinkedList example in Section 7.1, there are many method calls with the same target and source, which do not add any more information to the EOG. An implementation of such a mechanism would reduce the trace file size significantly.

It may also be possible to get rid of the intermediate step of the trace file and directly build up the EOG while tracing events. This may be done by either using the JNI inside the Tracing agent to call methods of the Type inferer, or by reimplementing the Tracing agent using JDI and incorporating it into the Type inferer. Both of these approaches have the disadvantage that only one test run at a time can be handled; the joining of multiple runs is not possible.

Threads The Inference tool does not work with multi-threaded programs yet. This is due to the fact that we had some problems with the thread ID that is supplied as argument to every callback function of the JVMTI agent. We have experienced sudden thread ID switches in the middle of a program execution. Some investigation has to be made on what causes these thread ID changes. Once the problem is resolved, the Type inferer can safely use one static call stack per thread ID and the whole program becomes thread safe.

Code coverage Until now, we have not investigated if general software engineering metrics measuring the test case quality apply well to our program. It is not clear whether a good test case in software engineering terms is also a good test case for our Inference tool. If so, it would bring the benefit that existing code coverage and testing tools can be used to automatically generate test cases as input for our tool.

Unannotated code In the current implementation, we simply use a default value for variables that were not annotated in the runtime inference phase. It may be possible to infer more variables by running a more intelligent global fixpoint iteration over all method bodies during the method body annotation. It also be possible to use the generated output from the harmonization and annotation visitor as input to the static inference tool by Kellenberger [16].

Indexing Producing valid indexes with the Inference tool has proved to be a difficult task. Since all instance initializers of a class are inlined into every constructor when the bytecode is generated, it is not possible to know where exactly the code of a constructor starts. This problem should probably be taken care of in the Annotation tool, because there, we parse source code information and could possibly detect the starting instruction of a constructor. The indexes for the constructors can then be split into indexes in instance initializers and indexes of the actual constructors.

The second indexing problem we still have is concerning local variables. As described in Section 5.5.3, a way has to find the index of local variable declarations within a method body has yet to be found. It should be possible, because the viewer tool of the bytecode library jclasslib [11] does display the local variables in the order they are declared within a method body. The question is whether this information is extractable with the BCEL library.

Superclass annotation The annotation of super declarations of methods is not supported yet. Methods declared in interfaces and methods of abstract classes are not annotated at all. Only fields of abstract classes are annotated correctly. Method entry and exit events are triggered by the actual class of an object, super classes or interfaces are not considered. Therefore, only the

actual class is annotated. We should be able to solve this problem, if one additional variable is added to the data structure for each super class or interface declaring that same method. This way, virtual calls to implementations of a method in subclasses can be harmonized.

7.4 Conclusion

The main contribution of this thesis is the extension of the Runtime Inference tool by array inference, static method handling, and method body inference. Whole Java programs can now be annotated with Universe types and the resulting annotations can be compiled with the Multijava compiler[7].

The abstract interpretation approach for method body annotation is able to deal with code that has not been visited by a given test case. This is used to infer Universe types for the signatures of methods that were never called. The user of the tool may provide default annotations where it is necessary. These default annotations will be considered only during the method body inference phase of the algorithm. Universe types inferred from the Extended Object Graph have precedence over these default types.

Furthermore, it is now possible to join several test runs, achieving better code coverage. Since code coverage is crucial for the quality of the resulting annotations, this possibility is very powerful. The dynamic structure of multiple runs is harmonized and mapped to Universe annotations of the Java source code.

The Inference tool is now practically applicable for small and medium programs. However, the handling of the trace file is the dominant factor for the execution time. The runtime of the Inference tool is virtually independent from the size of the source code (e.g. number of classes, lines of code, etc.). Nevertheless, the trace file can become much too big to handle within reasonable time if a test case is chosen that triggers many events.

Bibliography

[1] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 149–160. Springer-Verlag, 2004. Available from http://www.cs.sunysb.edu/~ragarwal/.

[2] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tampa Bay, FL, October 2001.

[3] Shigeru Chiba. javassist. Available from http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[4] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001. Available from http://www.cs.uu.nl/~dave/publications.html.

[5] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310. ACM Press, 2002.

[6] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.

[7] Craig Chambers Curtis Clifton, Gary T. Leavens and Todd Millstein. Multijava project. OOPSLA 2000, 2000. Available from http://multijava.sourceforge.net/.

[8] Lucy Mendel Derek Rayside and Daniel Jackson. A dynamic analysis for revealing object ownership and sharing. In *WODA*, 2006.

[9] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[10] T. Coupaye E. Bruneton, R. Lenglet. Asm: a code manipulation tool to implement adaptable systems. Technical report, France Télécom, R&D, November 2002. Available from http://asm.objectweb.org/current/asm-eng.pdf.

[11] ej technologies. Jclasslib. http://www.ej-technologies.com/products/jclasslib/overview.html.

[12] Etienne Gagnon. The SableVM Project. http://sablevm.org/.

[13] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 1996.

[14] David Graf. Implementing a purity and side effect analysis for java programs. Winter Semester 2005/2006.

[15] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

[16] Nathalie Kellenberger. Static runtime inference. Master's thesis, ETH Zurich, 2005.

[17] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979. Available from http://doi.acm.org/10.1145/357062.357071.

[18] Xavier Leroy. Java bytecode verification: An overview. *Lecture Notes in Computer Science*, 2102:265+, 2001.

[19] Frank Lyner. Runtime universe type inference. Master's thesis, ETH Zurich, 2005.

[20] Marco Meyer. Interaction with ownership graphs. January 2006.

[21] Samuel Moelius and Amie Souter. An object ownership inference algorithm and its applications. In *Proceedings of MASPLAS'04*. Mid-Atlantic Student Workshop on Programming Languages and Systems, April 2004. Available from http://sciris.shu.edu/masplas2004/proceedings.html.

[22] Matthias Niklaus. Static universe type inference using a sat-solver. Master's thesis, ETH Zurich, May 2006.

[23] Apache XML Project. Java xmlbeans, 2003. Available from http://xmlbeans.apache.org.

[24] Alexandru Sălcianu and Martin Rinard. A combined pointer and purity analysis for java programs. Master's thesis, Massachusetts Institute of Technology, 2004. Available from http://www.mit.edu/~salcianu/publications/vmcai05-purity.pdf.

[25] F. Yellin T. Lindholm. *The Java Virtual Machine Specification*. Java Series. Sun Microsystems, 1997.

[26] Tim Wilkinson. The Kaffe OpenVM. http://www.kaffe.org/.

[27] Alisdair Wren. Inferring ownership. Master's thesis, Imperial College, June 2003. Available from http://www.cl.cam.ac.uk/users/aw345/writings/.

Appendix A

State Transition Rules

This section states the stack and register transition rules for each bytecode instruction.

$$
\begin{aligned}
om(\tau) &= Ownership\ modifier\ of\ type\ \tau. \\
jt(\tau) &= Java\ type\ of\ type\ \tau. \\
tc(\tau_1, \tau_2) &= om(\tau_1) * om(\tau_2)\ jt(\tau_2) \\
comp(\alpha) &= Component\ type\ of\ array\ \alpha. \\
\tau <: \tau\prime &= \tau\ is\ a\ subtype\ of\ \tau\prime.\ (Ownership\ modifier\ included) \\
scs * (\tau_1, \tau_2) &= \begin{cases} \tau_2\ if\ \tau_1 = uninitialized \\ \tau_1\ if\ \tau_2 = uninitialized \\ smallest\_common\_supertype(\tau_1, \tau_2)\ if\ \tau_i \neq uninitialized \end{cases}
\end{aligned}
$$

a

aaload : $(\mathbf{int}.\alpha.S, R) \rightarrow (tc(\alpha, \tau).S, R),\ comp(\alpha) = \tau$
aastore : $(tc(\alpha, \tau).\mathbf{int}.\alpha.S, R) \rightarrow (S, R),\ comp(\alpha) = \tau \wedge om(\alpha) \neq readonly$
aconst_null : $(S, R) \rightarrow (\mathbf{null}.S, R)$
aload n : $(S, R) \rightarrow (R(n).S, R),\ R(n) <: Object$
aload_<n> : $(S, R) \rightarrow (R(n).S, R)$
anewarray α: $(\mathbf{int}.S, R) \rightarrow (\alpha.S, R)$
areturn : $(\tau.S, R) \rightarrow (\mathbf{empty}, R),\ \tau <: Object$
arraylength : $(\alpha.S, R) \rightarrow (\mathbf{int}.S, R)$
astore n : $(\tau.S, R) \rightarrow (S, R\{n \leftarrow scs * (\tau, R(n))\})$
astore_<n> : $(\tau.S, R) \rightarrow (S, R\{n \leftarrow scs * (\tau, R(n))\})$
athrow : $(\epsilon.S, R) \rightarrow (\epsilon, R),\ \epsilon <: Throwable$

b

baload : $(\mathbf{int}.\alpha.S, R) \rightarrow (\mathbf{int}.S, R),\ comp(\alpha) = \{\mathbf{byte}, \mathbf{boolean}\}$
bastore : $(\mathbf{int}.\mathbf{int}.\alpha.S, R) \rightarrow (S, R),\ comp(\alpha) = \{\mathbf{byte}, \mathbf{boolean}\} \wedge om(\alpha) \neq readonly$
bipush : $(S, R) \rightarrow (\mathbf{int}.S, R)$

c

caload : $(\mathbf{int}.\alpha.S, R) \rightarrow (\mathbf{int}.S, R),\ comp(\alpha) = \mathbf{char}$
castore : $(\mathbf{int}.\mathbf{int}.\alpha.S, R) \rightarrow (S, R),\ comp(\alpha) = \mathbf{char} \wedge om(\alpha) \neq readonly$

checkcast τ : $(\tau\prime.S, R) \rightarrow (\tau.S, R),\ om(\tau) = om(\tau\prime)$

d

d2f : $(\textbf{double}.S, R) \rightarrow (\textbf{float}.S, R)$
d2i : $(\textbf{double}.S, R) \rightarrow (\textbf{int}.S, R)$
d2l : $(\textbf{double}.S, R) \rightarrow (\textbf{long}.S, R)$
dadd : $(\textbf{double}.\textbf{double}.S, R) \rightarrow (\textbf{double}.S, R)$
daload : $(\textbf{int}.\alpha.S, R) \rightarrow (\textbf{double}.S, R),\ comp(\alpha) = \textbf{double}$
dastore : $(\textbf{double}.\textbf{int}.\alpha.S, R) \rightarrow (S, R),\ comp(\alpha) = \textbf{double} \wedge om(\alpha) \neq readonly$
dcmpg : $(\textbf{double}.\textbf{double}.S, R) \rightarrow (\textbf{int}.S, R)$
dcmpl : $(\textbf{double}.\textbf{double}.S, R) \rightarrow (\textbf{int}.S, R)$
dconst_<*d*> : $(S, R) \rightarrow (\textbf{double}.S, R)$
ddiv : $(\textbf{double}.\textbf{double}.S, R) \rightarrow (\textbf{double}.S, R)$
dload n : $(S, R) \rightarrow (\textbf{double}.S, R)R(n) = \textbf{double}$
dload_<*n*> : $(S, R) \rightarrow (\textbf{double}.S, R)R(n) = \textbf{double}$
dmul : $(\textbf{double}.\textbf{double}.S, R) \rightarrow (\textbf{double}.S, R)$
dneg : $(\textbf{double}.S, R) \rightarrow (\textbf{double}.S, R)$
drem : $(\textbf{double}.\textbf{double}.S, R) \rightarrow (\textbf{double}.S, R)$
dreturn : $(\textbf{double}.S, R) \rightarrow (\textbf{empty}, R)$
dstore n : $(\textbf{double}.S, R) \rightarrow (S, R\{n \leftarrow \textbf{double}\})$
dstore_<*n*> : $(\textbf{double}.S, R) \rightarrow (S, R\{n \leftarrow \textbf{double}\})$
dsub : $(\textbf{double}.\textbf{double}.S, R) \rightarrow (\textbf{double}.S, R)$
dup : $(\tau.S, R) \rightarrow (\tau.\tau.S, R),\ \tau \neq \{\textbf{long}, \textbf{double}\}$
dup_x1 : $(\tau_1.\tau_2.S, R) \rightarrow (\tau_1.\tau_2.\tau_1.S, R),\ \tau_i \neq \{\textbf{long}, \textbf{double}\}$

dup_x2 : $(\tau_1.\tau_2.\tau_3.S, R) \rightarrow (\tau_1.\tau_2.\tau_3.\tau_1.S, R)\ if\ \tau_x \neq \{\textbf{long, double}\}$
$(\tau_1.\tau_2.S, R) \rightarrow (\tau_1.\tau_2.\tau_1.S, R)\ if\ \tau_2 \neq \textbf{long}, \textbf{double} \wedge \tau_1 = \{\textbf{long, double}\}$

dup2 : $(\tau_1.\tau_2.S, R) \rightarrow (\tau_1.\tau_2.\tau_1.\tau_2.S, R)\ if\ \tau_x \neq \{\textbf{long}, \textbf{double}\}$
$(\tau.S, R) \rightarrow (\tau.\tau.S, R)\ if\ \tau = \{\textbf{long}, \textbf{double}\}$

dup2_x1: $(\tau_1.\tau_2.\tau_3.S, R) \rightarrow (\tau_1.\tau_2.\tau_3.\tau_1.\tau_2.S, R)\ if\ \tau_x \neq \{\textbf{long}, \textbf{double}\}$
$(\tau_1.\tau_2.S, R) \rightarrow (\tau_1.\tau_2.\tau_1.S, R)\ if\ \tau_1 = \{\textbf{long}, \textbf{double}\} \wedge \tau_2 \neq \{\textbf{long}, \textbf{double}\}$

dup2_x2: $(\tau_1.\tau_2.\tau_3.\tau_4.S, R) \rightarrow (\tau_1.\tau_2.\tau_3.\tau_4.\tau_1.\tau_2.S, R)\ if\ \tau_x \neq \{\textbf{long}, \textbf{double}\}$
$(\tau_1.\tau_2.\tau_3.S, R) \rightarrow (\tau_1.\tau_2.\tau_3.\tau_1.S, R)\ if\ \tau_1 = \{\textbf{long}, \textbf{double}\} \wedge \tau_2, \tau_3 \neq \{\textbf{long}, \textbf{double}\}$
$(\tau_1.\tau_2.\tau_3.S, R) \rightarrow (\tau_1.\tau_2.\tau_3.\tau_1.\tau_2.S, R)\ if\ \tau_3 = \{\textbf{long}, \textbf{double}\} \wedge \tau_1, \tau_2 \neq \{\textbf{long}, \textbf{double}\}$
$(\tau_1.\tau_2.S, R) \rightarrow (\tau_1.\tau_2.\tau_1.S, R)\ if\ \tau_x = \{\textbf{long}, \textbf{double}\}$

f

f2d : $(\textbf{float}.S, R) \rightarrow (\textbf{double}.S, R)$
f2i : $(\textbf{float}.S, R) \rightarrow (\textbf{int}.S, R)$
f2l : $(\textbf{float}.S, R) \rightarrow (\textbf{long}.S, R)$
fadd : $(\textbf{float}.\textbf{float}.S, R) \rightarrow (\textbf{float}.S, R)$
faload : $(\textbf{int}.\alpha.S, R) \rightarrow (\textbf{float}.S, R),\ comp(\alpha) = \textbf{float}$
fastore : $(\textbf{float}.\textbf{int}.\alpha.S, R) \rightarrow (S, R),\ comp(\alpha) = \textbf{float} \wedge om(\alpha) \neq readonly$
fcmpg : $(\textbf{float}.\textbf{float}.S, R) \rightarrow (\textbf{int}.S, R)$
fcmpl : $(\textbf{float}.\textbf{float}.S, R) \rightarrow (\textbf{int}.S, R)$
fconst_<*d*> : $(S, R) \rightarrow (\textbf{float}.S, R)$
fdiv : $(\textbf{float}.\textbf{float}.S, R) \rightarrow (\textbf{float}.S, R)$
fload n : $(S, R) \rightarrow (\textbf{float}.S, R), R(n) = \textbf{float}$

fload_<*n*> : $(S, R) \rightarrow (\mathbf{float}.S, R), R(n) = \mathbf{float}$
fmul : $(\mathbf{float}.\mathbf{float}.S, R) \rightarrow (\mathbf{float}.S, R)$
fneg : $(\mathbf{float}.S, R) \rightarrow (\mathbf{float}.S, R)$
frem : $(\mathbf{float}.\mathbf{float}.S, R) \rightarrow (\mathbf{float}.S, R)$
freturn : $(\mathbf{float}.S, R) \rightarrow (\mathbf{empty}, R)$
fstore n : $(\mathbf{float}.S, R) \rightarrow (S, R\{n \leftarrow \mathbf{float}\})$
fstore_<*n*> : $(\mathbf{float}.S, R) \rightarrow (S, R\{n \leftarrow \mathbf{float}\})$
fsub : $(\mathbf{float}.\mathbf{float}.S, R) \rightarrow (\mathbf{float}.S, R)$

g

getfield $C.f.\tau$: $(\tau\prime.S, R) \rightarrow (tc(\tau\prime, \tau).S, R)$ *if* $\tau\prime <: C$
getstatic $C.f.\tau$: $(S, R) \rightarrow (\tau.S, R)$
goto : $(S, R) \rightarrow (S, R)$
goto_w : $(S, R) \rightarrow (S, R)$

i

i2b : $(\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
i2c : $(\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
i2d : $(\mathbf{int}.S, R) \rightarrow (\mathbf{double}.S, R)$
i2f : $(\mathbf{int}.S, R) \rightarrow (\mathbf{float}.S, R)$
i2l : $(\mathbf{int}.S, R) \rightarrow (\mathbf{long}.S, R)$
i2s : $(\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
iadd : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
iaload : $(\mathbf{int}.\alpha.S, R) \rightarrow (\mathbf{int}.S, R),\ comp(\alpha) = \mathbf{int}$
iand : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
iastore : $(\mathbf{int}.\mathbf{int}.\alpha.S, R) \rightarrow (S, R),\ comp(\alpha) = \mathbf{int} \wedge om(\alpha) \neq readonly$
iconst_<*i*> : $(S, R) \rightarrow (\mathbf{int}.S, R)$
idiv : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
if_acmp<cond> : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (S, R)$
if_icmp<cond> : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (S, R)$
if<cond> : $(\mathbf{int}.S, R) \rightarrow (S, R)$
ifnonnull : $(\tau.S, R) \rightarrow (S, R)$
ifnull : $(\tau.S, R) \rightarrow (S, R)$
iinc : $(S, R) \rightarrow (S, R)$
iload n : $(S, R) \rightarrow (\mathbf{int}.S, R),\ R(n) = \mathbf{int}$
iload_<*n*> : $(S, R) \rightarrow (\mathbf{int}.S, R),\ R(n) = \mathbf{int}$
imul : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
ineg : $(\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
instanceof : $(\tau.S, R) \rightarrow (\mathbf{int}.S, R)$
invokeinterface C.m.σ : $(tc(\tau\prime, \tau_n\prime)...tc(\tau\prime, \tau_1\prime).\tau\prime.S, R) \rightarrow (tc(\tau\prime, \tau).S, R)$
if $\sigma = \tau(\tau_1, ..., \tau_n),\ \tau\prime <: C,\ \tau_i\prime <: \tau_i$ *for* $i = 1...n,\ om(\tau\prime) \neq readonly,$
if $\exists\ \tau_i$ *with* $om(\tau_i) = rep$ *then* $om(\tau\prime) = peer\_this$
invokespecial C.m.σ : $(tc(\tau\prime, \tau_n\prime)...tc(\tau\prime, \tau_1\prime).\tau\prime.S, R) \rightarrow (tc(\tau\prime, \tau).S, R)$
if $\sigma = \tau(\tau_1, ..., \tau_n),\ \tau\prime <: C,\ \tau_i\prime <: \tau_i$ *for* $i = 1...n,\ om(\tau\prime) \neq readonly,$
if $\exists\ \tau_i$ *with* $om(\tau_i) = rep$ *then* $om(\tau\prime) = peer\_this$
invokestatic ι C.m.σ : $(tc(\iota, \tau_n\prime)...tc(\iota, \tau_1\prime).S, R) \rightarrow (tc(\iota, \tau).S, R)$
if $\sigma = \tau(\tau_1, ..., \tau_n),\ \tau_i\prime <: \tau_i$ *for* $i = 1...n,\ \iota = invocation\ type$
invokevirtual C.m.σ : $(tc(\tau\prime, \tau_n\prime)...tc(\tau\prime, \tau_1\prime).\tau\prime.S, R) \rightarrow (tc(\tau\prime, \tau).S, R)$
if $\sigma = \tau(\tau_1, ..., \tau_n),\ \tau\prime <: C,\ \tau_i\prime <: \tau_i$ *for* $i = 1...n,\ om(\tau\prime) \neq readonly,$
if $\exists\ \tau_i$ *with* $om(\tau_i) = rep$ *then* $om(\tau\prime) = peer\_this$

ior : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
irem : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
ireturn : $(\mathbf{int}.S, R) \rightarrow (\mathbf{empty}, R)$
ishl : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
ishr : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
istore n : $(\mathbf{int}.S, R) \rightarrow (S, R\{n \leftarrow \mathbf{int}\})$
istore_<n> : $(\mathbf{int}.S, R) \rightarrow (S, R\{n \leftarrow \mathbf{int}\})$
isub : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
iushr : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$
ixor : $(\mathbf{int}.\mathbf{int}.S, R) \rightarrow (\mathbf{int}.S, R)$

j

jsr : $(S, R) \rightarrow (\mathbf{returnAddress}.S, R)$
jsr_w : $(S, R) \rightarrow (\mathbf{returnAddress}.S, R)$

l

l2d : $(\mathbf{long}.S, R) \rightarrow (\mathbf{double}.S, R)$
l2f : $(\mathbf{long}.S, R) \rightarrow (\mathbf{float}.S, R)$
l2i : $(\mathbf{long}.S, R) \rightarrow (\mathbf{int}.S, R)$
ladd : $(\mathbf{long}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$
laload : $(\mathbf{int}.\alpha.S, R) \rightarrow (\mathbf{long}.S, R),\ comp(\alpha) = \mathbf{long}$
land : $(\mathbf{long}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$
lastore : $(\mathbf{long}.\mathbf{int}.\alpha.S, R) \rightarrow (S, R),\ comp(\alpha) = \mathbf{long} \wedge om(\alpha) \neq readonly$
lcmp : $(\mathbf{long}.\mathbf{long}.S, R) \rightarrow (\mathbf{int}.S, R)$
lconst_<l> : $(S, R) \rightarrow (\mathbf{long}.S, R)$

ldc *index* : $(S, R) \rightarrow (\mathbf{int}.S, R)\ if\ cpool(index) = \mathbf{int}$
$(S, R) \rightarrow (\mathbf{float}.S, R)\ if\ cpool(index) = \mathbf{float}$
$(S, R) \rightarrow (\tau.S, R)\ if\ cpool(index) = \tau,\ with\ \tau <: String$

ldc_w *index* : $(S, R) \rightarrow (\mathbf{int}.S, R)\ if\ cpool(index) = \mathbf{int}$
$(S, R) \rightarrow (\mathbf{float}.S, R)\ if\ cpool(index) = \mathbf{float}$
$(S, R) \rightarrow (\tau.S, R)\ if\ cpool(index) = \tau,\ with\ \tau <: String$

ldc2_w *index* : $(S, R) \rightarrow (\mathbf{long}.S, R)\ if\ cpool(index) = \mathbf{long}$
$(S, R) \rightarrow (\mathbf{double}.S, R)\ if\ cpool(index) = \mathbf{double}$

ldiv : $(\mathbf{long}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$
lload n : $(S, R) \rightarrow (\mathbf{long}.S, R),\ R(n) = \mathbf{long}$
lload_<n> : $(S, R) \rightarrow (\mathbf{long}.S, R),\ R(n) = \mathbf{long}$
lmul : $(\mathbf{long}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$
lneg : $(\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$
lookupswitch : $(\mathbf{int}.S, R) \rightarrow (S, R)$
lor : $(\mathbf{long}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$
lrem : $(\mathbf{long}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$
lreturn : $(\mathbf{long}.S, R) \rightarrow (\mathbf{empty}, R)$
lshl : $(\mathbf{int}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$

lshr : $(\mathbf{int}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$
lstore n : $(\mathbf{long}.S, R) \rightarrow (S, R\{n \leftarrow \mathbf{long}\})$
lstore_<n> : $(\mathbf{long}.S, R) \rightarrow (S, R\{n \leftarrow \mathbf{long}\})$
lsub : $(\mathbf{long}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$
lushr : $(\mathbf{int}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$
lxor : $(\mathbf{long}.\mathbf{long}.S, R) \rightarrow (\mathbf{long}.S, R)$

m

monitorenter : $(\tau.S, R) \rightarrow (S, R)$
monitorexit : $(\tau.S, R) \rightarrow (S, R)$
multianewarray ι C n : $(\mathbf{int_1}...\mathbf{int_n}.S, R) \rightarrow (\alpha.S, R),\ n \geq 1,\ \alpha <: C,\ om(\alpha) = \iota,$
$\iota = invocation\ type$

n

new ι C : $(S, R) \rightarrow (\tau.S, R),\ \tau <: C,\ om(\tau) = \iota,\ \iota = invocation\ type$
newarray ι *atype* : $(\mathbf{int}.S, R) \rightarrow (\alpha.S, R),\ comp(\alpha) = atype,\ om(\alpha) = \iota,\ \iota = invocation\ type$
nop : $(S, R) \rightarrow (S, R)$

p

pop : $(\tau.S, R) \rightarrow (S, R),\ \tau \neq \{\mathbf{long}, \mathbf{double}\}$

pop2 : $(\tau_1.\tau_2.S, R) \rightarrow (S, R)\ if\ \tau_i \neq \{\mathbf{long}, \mathbf{double}\}$
$(\tau.S, R) \rightarrow (S, R)\ if\ \tau = \{\mathbf{long}, \mathbf{double}\}$

putfield C.f.τ : $(tc(\tau_2, \tau_1).\tau_2.S, R) \rightarrow (S, R)\ if\ \tau_1 <: \tau, \tau_2 <: C, om(\tau_2) \neq readonly$
putstatic C.f.τ : $(\tau\prime.S, R) \rightarrow (S, R),\ \tau\prime <: \tau,\ om(\tau) = readonly$

r

ret n : $(S, R) \rightarrow (S, R),\ R(n) = \mathbf{returnAddress}$
return : $(S, R) \rightarrow (\mathbf{empty}, R)$

s

saload : $(\mathbf{int}.\alpha.S, R) \rightarrow (\mathbf{int}.S, R),\ comp(\alpha) = \mathbf{short}$
sastore : $(\mathbf{int}.\mathbf{int}.\alpha.S, R) \rightarrow (S, R),\ comp(\alpha) = \mathbf{short} \wedge om(\alpha) \neq readonly$
sipush : $(S, R) \rightarrow (\mathbf{int}.S, R)$
swap : $(\tau_1.\tau_2.S, R) \rightarrow (\tau_2.\tau_1.S, R),\ \tau_i \neq \{\mathbf{double}, \mathbf{long}\}$

t

tableswitch : $(\mathbf{int}.S, R) \rightarrow (S, R)$

w

wide : $(S, R) \rightarrow (S, R)$ (no change on stack, only the behavior of following instruction is changed)

Appendix B

Tree Example

Collection

```
1  interface Collection {
2          public Iterator elements();
3
4          public void addElement(Integer i);
5
6          public boolean isEmpty();
7  }
```

ArrayCollection

```
1  public class ArrayCollection implements Collection {
2
3          public rep readonly Integer [] data;
4          public int pos;
5          private static final int SIZE = 64;
6
7          public ArrayCollection () {
8                  data = new rep readonly Integer[SIZE];
9                  pos = 0;
10         }
11
12         public void addElement(Integer i) {
13                 if(pos >= data.length) {
14                         rep readonly Integer [] oldData = data;
15                         data = new rep readonly Integer[data.length*2];
16                         for(int j = 0; j<oldData.length; j++) {
17                                 data[j] = oldData[j];
18                         }
19                         data[pos++] = i;
20                 }
21         }
22
23         public boolean isEmpty() {
24                 return (pos == 0);
25         }
26
27         public peer Iterator elements() {
28                 return new peer ArrayCollectionIterator (this);
```

```
29          }
30  }
```

SortedTree

```
1   public class SortedTree implements Collection {
2           protected rep SortedTreeNode rootNode;
3
4           public SortedTree() {
5                   rootNode = null;
6           }
7
8           public boolean isEmpty() {
9                   return (rootNode == null);
10          }
11
12          public void addElement(Integer i) {
13                  if (rootNode == null) {
14                          rootNode = new rep SortedTreeNode(i);
15                  }
16                  else if (i.intValue() <=rootNode.key.intValue()) {
17                          insertInLeftSubtree (i, rootNode);
18                  }
19                  else {
20                          insertInRightSubtree (i, rootNode);
21                  }
22          }
23
24          public peer Iterator elements() {
25                  return peer createInorderIterator ();
26          }
27
28          public rep Iterator createPostorderIterator () {
29                  return new rep PostorderTreeIterator(rootNode);
30          }
31
32          public peer Iterator createInorderIterator () {
33                  return new peer InorderTreeIterator(rootNode);
34          }
35
36          private void insertInLeftSubtree (Integer i, rep SortedTreeNode node) {
37                  rep SortedTreeNode leftChild = node.left;
38                  if (leftChild == null) {
39                          rep SortedTreeNode newnode = new rep SortedTreeNode(i, node);
40                          node.left = newnode;
41                  }
42                  else if (i.intValue() <= leftChild.key.intValue()) {
43                          insertInLeftSubtree (i, leftChild );
44                  }
45                  else {
46                          insertInRightSubtree (i, leftChild );
47                  }
48          }
49
50          private void insertInRightSubtree (Integer i, rep SortedTreeNode node) {
51                  rep SortedTreeNode rightChild = node.right;
52                  if (rightChild == null) {
```

```
53                          rep SortedTreeNode newnode = new rep SortedTreeNode(i, node);
54                          node.right = newnode;
55                  }
56                  else if (i.intValue() <= rightChild.key.intValue()) {
57                          insertInLeftSubtree (i, rightChild );
58                  }
59                  else {
60                          insertInRightSubtree (i, rightChild );
61                  }
62          }
63  }
```

SortedTreeNode

```
1   public class SortedTreeNode {
2
3           public Integer key;
4
5           public peer SortedTreeNode left;
6
7           public peer SortedTreeNode right;
8
9           public peer SortedTreeNode parent;
10
11          public SortedTreeNode(Integer i) {
12                  key = i;
13                  this.parent = null;
14          }
15
16          public SortedTreeNode(Integer i, peer SortedTreeNode parent) {
17                  key = i;
18                  this.parent = parent;
19          }
20  }
```

Iterator

```
1   interface Iterator {
2
3           public void first ();
4
5           public void next() throws NoSuchElementException;
6
7           public Integer getCurrent();
8
9           public boolean isDone();
10  }
```

ArrayCollectionIterator

```
1   public class ArrayCollectionIterator implements Iterator {
2           private int pos;
3           private peer ArrayCollection coll;
4
```

```
5           public  ArrayCollectionIterator (peer ArrayCollection  coll ) {
6                   this . coll  = coll ;
7           }
8
9           public void  first () {
10                  pos = 0;
11          }
12
13          public void next() throws NoSuchElementException {
14                  if (pos < coll . data . length  -1) {
15                          pos++;
16                  }
17                  else {
18                          throw new rep NoSuchElementException("ArrayCollectionIterator");
19                  }
20          }
21
22          public Integer  getCurrent() {
23                  return  coll . data[pos];
24          }
25
26          public boolean isDone() {
27                  return (pos >= coll.pos) || (pos >= coll.data . length );
28          }
29  }
```

TreeIterator

```
1   abstract class  TreeIterator  implements Iterator {
2
3           protected readonly SortedTreeNode currentNode;
4
5           protected readonly SortedTreeNode startNode;
6
7           public void  first () {
8                   currentNode = startNode;
9           }
10
11          abstract public void next() throws NoSuchElementException;
12
13          public Integer  getCurrent() {
14                  return currentNode.key;
15          }
16
17          public boolean isDone() {
18                  return (currentNode == null);
19          }
20  }
```

InorderTreeIterator

```
1   public class  InorderTreeIterator  extends TreeIterator  {
2
3           public  InorderTreeIterator (readonly SortedTreeNode rootNode) {
4                   startNode = rootNode;
```

```
5                   if (rootNode != null) {
6                           while(startNode. left  != null) {
7                                   startNode = startNode. left ;
8                           }
9                   }
10                  currentNode = startNode;
11          }
12
13          public void next() throws NoSuchElementException {
14                  if (currentNode == null) {
15                          throw new rep NoSuchElementException("InorderTreeIterator");
16                  }
17                  else if (currentNode. right  != null) {
18                          currentNode = currentNode.right;
19                          while(currentNode. left  != null) {
20                                  currentNode = currentNode.left;
21                          }
22                  }
23                  else {
24                          while((currentNode.parent  != null) && (currentNode.parent. left  != currentNode)) {
25
26                                  currentNode = currentNode.parent;
27                          }
28                          currentNode = currentNode.parent;
29                  }
30          }
31  }
```

PostorderTreeIterator

```
1   public class  PostorderTreeIterator  extends  TreeIterator  {
2
3           public  PostorderTreeIterator (readonly SortedTreeNode rootNode) {
4                   startNode = rootNode;
5                   if (startNode != null) {
6                           while((startNode. left  != null)  ||  (startNode. right  != null)) {
7                                   while(startNode. right  != null) {
8                                           startNode = startNode. right ;
9                                   }
10                                  if (startNode. left  != null) {
11                                          startNode = startNode. left ;
12                                  }
13                          }
14                  }
15                  currentNode = startNode;
16          }
17
18          public void next() throws NoSuchElementException {
19                  if (currentNode == null) {
20                          throw new NoSuchElementException("PostorderTreeIterator");
21                  }
22                  if (currentNode.parent  == null) {
23                          currentNode = currentNode.parent;
24                          return;
25                  }
26                  if (currentNode.parent. right  == currentNode) {
27                          if (currentNode.parent. left  != null) {
```

```
28                              currentNode = currentNode.parent. left ;
29                      }
30                      else {
31                              currentNode = currentNode.parent;
32                              return;
33                      }
34                      while((currentNode. left  != null)  ||  (currentNode. right  != null)) {
35                              while(currentNode. right  != null)  {
36                                      currentNode = currentNode.right;
37                              }
38                              if (currentNode. left  != null)  {
39                                      currentNode = currentNode.left ;
40                              }
41                      }
42              }
43              else {
44                      currentNode = currentNode.parent;
45              }
46      }
47  }
```

Statistics

```
1   interface  Statistics  {
2           public void sample(Integer  i );
3
4           public Object  result ();
5
6           public String  getName();
7   }
```

SumAndSubtract

```
1   public  class SumAndSubtract implements Statistics {
2
3           private  int  sum;
4           private  int  operator ;
5
6           public SumAndSubtract() {
7                   sum = 0;
8                   operator  = 1;
9           }
10
11          public void sample(Integer  i ) {
12                  sum += operator * i.intValue ();
13                  operator  -= operator;
14          }
15
16          public Object  result () {
17                  return new Integer(sum);
18          }
19
20          public String  getName() {
21                  return "SumAndSubtract";
22          }
```

```
23 }
```

Average

```
1  public class Average implements Statistics {
2
3          private int sum;
4          private int counter;
5
6          public Average() {
7                  sum = 0;
8                  counter = 0;
9          }
10
11         public void sample(Integer i) {
12                 sum += i.intValue();
13                 counter++;
14         }
15
16         public Object result () {
17                 return new Double(sum / (double) counter);
18         }
19
20         public String getName() {
21                 return "Average";
22         }
23 }
```

Checker

```
1  public class Checker implements Statistics {
2
3          private boolean found;
4          private Integer element;
5
6          public Checker(Integer val) {
7                  element = val;
8                  found = false;
9          }
10
11         public void sample(Integer i) {
12                 if (i.equals(element)) {
13                         found = true;
14                 }
15         }
16
17         public Object result () {
18                 return new Boolean(found);
19         }
20
21         public String getName() {
22                 return "Checker";
23         }
24 }
```

Appendix C

Agentoutput XML Schema

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="trace">
4      <xs:complexType>
5        <xs:sequence>
6                  <xs:element ref="classPrepare" minOccurs="0" maxOccurs="unbounded"/>
7                  <xs:element ref="methodentry" minOccurs="0" maxOccurs="unbounded"/>
8                  <xs:element ref="methodexit" minOccurs="0" maxOccurs="unbounded"/>
9                  <xs:element ref=" fieldmodification " minOccurs="0" maxOccurs="unbounded"/>
10       </xs:sequence>
11     </xs:complexType>
12   </xs:element>
13 <!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->
14 <!-- class prepare event -->
15   <xs:element name="classPrepare">
16     <xs:complexType>
17       <xs:sequence>
18         <xs:element ref="field" minOccurs="0" maxOccurs="unbounded" />
19         <xs:element ref="method" minOccurs="0" maxOccurs="unbounded" />
20       </xs:sequence>
21       <xs:attribute name="name" type="xs:string"/>
22       <xs:attribute name="id" type="xs:int" />
23     </xs:complexType>
24   </xs:element>
25 <!-- field -->
26   <xs:element name="field">
27     <xs:complexType>
28       <xs:attribute name="id" type="xs:int" use="required" />
29       <xs:attribute name="name" type="xs:string"/>
30     </xs:complexType>
31   </xs:element>
32 <!-- method -->
33   <xs:element name="method">
34     <xs:complexType>
35       <xs:attribute name="id" type="xs:int" use="required" />
36       <xs:attribute name="name" type="xs:string"/>
37       <xs:attribute name="signature" type="xs:string "/>
38     </xs:complexType>
39   </xs:element>
40 <!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->
41 <!-- method entry event -->
42   <xs:element name="methodentry">
```

```
43      <xs:complexType>
44        <xs:sequence>
45          <xs:element ref="method" minOccurs="1" maxOccurs="1" />
46          <xs:element ref="callerObj" minOccurs="1" maxOccurs="1" />
47          <xs:element ref="targetObj" minOccurs="1" maxOccurs="1" />
48          <xs:element ref="callingMethod" minOccurs="1" maxOccurs="1" />
49          <xs:element ref="location" minOccurs="1" maxOccurs="1" />
50          <xs:element ref="param" minOccurs="0" maxOccurs="unbounded" />
51        </xs:sequence>
52      </xs:complexType>
53    </xs:element>
54  <!-- callerObj -->
55    <xs:element name="callerObj">
56      <xs:complexType>
57        <xs:attribute name="id" type="xs:int" use="required" />
58      </xs:complexType>
59    </xs:element>
60  <!-- targetObj -->
61    <xs:element name="targetObj">
62      <xs:complexType>
63        <xs:attribute name="id" type="xs:int" use="required" />
64      </xs:complexType>
65    </xs:element>
66  <!-- callingMethod -->
67    <xs:element name="callingMethod">
68      <xs:complexType>
69        <xs:attribute name="id" type="xs:int" use="required" />
70      </xs:complexType>
71    </xs:element>
72  <!-- location -->
73    <xs:element name="location">
74      <xs:complexType>
75        <xs:attribute name="id" type="xs:int" use="required" />
76      </xs:complexType>
77    </xs:element>
78  <!-- param -->
79    <xs:element name="param">
80      <xs:complexType>
81        <xs:attribute name="id" type="xs:int" use="required" />
82        <xs:attribute name="value" type="xs:int" use="required" />
83      </xs:complexType>
84    </xs:element>
85  <!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->
86  <!-- method exit event -->
87    <xs:element name="methodexit">
88      <xs:complexType>
89        <xs:sequence>
90          <xs:element ref="method" minOccurs="1" maxOccurs="1" />
91          <xs:element ref="targetObj" minOccurs="1" maxOccurs="1" />
92          <xs:element ref="returnValue" minOccurs="1" maxOccurs="1" />
93        </xs:sequence>
94      </xs:complexType>
95    </xs:element>
96  <!-- returnValue -->
97    <xs:element name="returnValue">
98      <xs:complexType>
99        <xs:attribute name="id" type="xs:int" use="required" />
100     </xs:complexType>
```

```
101     </xs:element>
102   <!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->
103   <!-- field modification event -->
104     <xs:element name="fieldmodification">
105       <xs:complexType>
106         <xs:sequence>
107           <xs:element ref="field" minOccurs="1" maxOccurs="1" />
108           <xs:element ref="modifiedObj" minOccurs="1" maxOccurs="1" />
109           <xs:element ref="modifyingObj" minOccurs="1" maxOccurs="1" />
110           <xs:element ref="referencedObj" minOccurs="1" maxOccurs="1" />
111         </xs:sequence>
112       </xs:complexType>
113     </xs:element>
114   <!-- modifiedObj -->
115     <xs:element name="modifiedObj">
116       <xs:complexType>
117         <xs:attribute name="id" type="xs:int" use="required" />
118       </xs:complexType>
119     </xs:element>
120   <!-- modifyingObj -->
121     <xs:element name="modifyingObj">
122       <xs:complexType>
123         <xs:attribute name="id" type="xs:int" use="required" />
124       </xs:complexType>
125     </xs:element>
126   <!-- referencedObj -->
127     <xs:element name="referencedObj">
128       <xs:complexType>
129         <xs:attribute name="id" type="xs:int" use="required" />
130       </xs:complexType>
131     </xs:element>
132   <!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->
133   <!-- array created event -->
134     <xs:element name="arraycreated">
135       <xs:complexType>
136         <xs:sequence>
137           <!-- Indicates that this array is multidimensional -->
138           <xs:element ref="multidim" minOccurs="0" maxOccurs="1" />
139           <xs:element ref="creatorObj" minOccurs="1" maxOccurs="1" />
140           <xs:element ref="createdObj" minOccurs="1" maxOccurs="1" />
141           <xs:element ref="subarray" minOccurs="0" maxOccurs="unbounded" />
142         </xs:sequence>
143       </xs:complexType>
144     </xs:element>
145   <!-- multidim -->
146     <xs:element name="multidim" >
147       <xs:complexType>
148       </xs:complexType>
149     </xs:element>
150   <!-- creatorObj -->
151     <xs:element name="creatorObj">
152       <xs:complexType>
153         <xs:attribute name="id" type="xs:int" use="required" />
154       </xs:complexType>
155     </xs:element>
156   <!-- createdObj -->
157     <xs:element name="createdObj">
158       <xs:complexType>
```

```
159         <xs:attribute name="id" type="xs:int" use="required" />
160       </xs:complexType>
161     </xs:element>
162   <!-- subarray -->
163     <xs:element name="subarray">
164       <xs:complexType>
165         <xs:attribute name="id" type="xs:int" use="required" />
166       </xs:complexType>
167     </xs:element>
168   <!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->
169   <!-- array component updated event -->
170     <xs:element name="arraycompupdate">
171       <xs:complexType>
172         <xs:sequence>
173           <!-- indicates that this is an object array -->
174           <xs:element ref="objectarray" minOccurs="0" maxOccurs="1" />
175           <!-- indicates that the value assigned is a subarray -->
176           <xs:element ref="subarrayvalue" minOccurs="0" maxOccurs="1" />
177           <xs:element ref="callerObj" minOccurs="1" maxOccurs="1" />
178           <xs:element ref="arrayObj" minOccurs="1" maxOccurs="1" />
179           <xs:element ref="valueObj" minOccurs="1" maxOccurs="1" />
180         </xs:sequence>
181       </xs:complexType>
182     </xs:element>
183   <!-- objectarray -->
184     <xs:element name="objectarray" >
185       <xs:complexType>
186       </xs:complexType>
187     </xs:element>
188   <!-- subarrayvalue -->
189     <xs:element name="subarrayvalue" >
190       <xs:complexType>
191       </xs:complexType>
192     </xs:element>
193   <!-- arrayObj -->
194     <xs:element name="arrayObj">
195       <xs:complexType>
196         <xs:attribute name="id" type="xs:int" use="required" />
197       </xs:complexType>
198     </xs:element>
199   <!-- valueObj -->
200     <xs:element name="valueObj">
201       <xs:complexType>
202         <xs:attribute name="id" type="xs:int" use="required" />
203       </xs:complexType>
204     </xs:element>
205   <!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->
206   </xs:schema>
```

Appendix D

Annotations XML Schema

```
1  <?xml version="1.0"?>
2
3  <!--
4  Schema for annotation files that specify Universe annotations that
5  should be added to existing sources.
6
7  Author: WMD
8  -->
9
10 <!--
11 TODO:
12 -
13 -->
14
15 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:po="http://sct.inf.ethz.ch/↘
   →annotations" targetNamespace="http://sct.inf.ethz.ch/annotations">
16
17
18 <!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
19 Some additional types to automatically check the input.
20 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->
21
22 <!--
23 The modifiers that are valid for simple reference types.
24 -->
25 <xsd:simpleType name="SimpleUniverseModifier">
26   < xsd:restriction base="xsd:string ">
27     <xsd:enumeration value="implicit_peer "/>
28     <xsd:enumeration value="peer"/>
29     <xsd:enumeration value="rep"/>
30     <xsd:enumeration value="readonly"/>
31   </ xsd:restriction >
32 </xsd:simpleType>
33
34 <!--
35 The modifiers that are valid for types, including arrays.
36 -->
37 <xsd:simpleType name="UniverseModifier">
38   < xsd:restriction base="xsd:string ">
39     <xsd:enumeration value="implicit_peer "/>
40     <xsd:enumeration value="peer"/>
41     <xsd:enumeration value="rep"/>
```

```
42      <xsd:enumeration value="readonly"/>
43
44      <xsd:enumeration value="peer␣peer"/>
45      <xsd:enumeration value="peer␣readonly"/>
46      <xsd:enumeration value="rep␣peer"/>
47      <xsd:enumeration value="rep␣readonly"/>
48      <xsd:enumeration value="readonly␣peer"/>
49      <xsd:enumeration value="readonly␣readonly"/>
50    </ xsd:restriction >
51  </xsd:simpleType>
52
53
54  <!--
55  The modifiers that are valid for methods.
56  -->
57  <xsd:simpleType name="UniverseMethodModifier">
58    < xsd:restriction  base="xsd:string ">
59      <xsd:enumeration value=""/>
60      <xsd:enumeration value="pure"/>
61    </ xsd:restriction >
62  </xsd:simpleType>
63
64  <xsd:simpleType name="CastPositionType">
65    < xsd:restriction  base="xsd:string ">
66      <xsd:enumeration value="assignment"/>
67      <xsd:enumeration value="method_call"/>
68      <xsd:enumeration value=" array_initialyzer "/>
69    </ xsd:restriction >
70  </xsd:simpleType>
71
72
73  <!--
74  What target should be modified?
75  -->
76  <xsd:simpleType name="ToolTarget">
77    < xsd:restriction  base="xsd:string ">
78      <!-- Modify the original Java sources -->
79      <xsd:enumeration value="java"/>
80
81      <!-- Create JML specification files -->
82      <xsd:enumeration value="jml"/>
83    </ xsd:restriction >
84  </xsd:simpleType>
85
86
87  <!--
88  With what style should the annotations be inserted ?
89  -->
90  <xsd:simpleType name="ToolStyle">
91    < xsd:restriction  base="xsd:string ">
92      <!-- As standard type annotations, e.g. "peer␣T" -->
93      <xsd:enumeration value="types"/>
94
95      <!-- Within JML comments, e.g. "/*@␣peer␣T␣@*/" -->
96      <xsd:enumeration value="jml"/>
97
98      <!-- As escaped JML comments, e.g. "/*@␣\peer␣T␣@*/" -->
99      <xsd:enumeration value="oldjml"/>
```

```
100     </ xsd:restriction >
101   </xsd:simpleType>
102
103
104   <!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
105   The elements of our schema.
106   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->
107
108   <!--
109   The top-level element consisting of one header element and
110   at least one class element.
111   -->
112   <xsd:element name="annotations">
113     <xsd:complexType>
114       <xsd:sequence>
115         <xsd:element ref="po:head" minOccurs="1" maxOccurs="1"/>
116         <xsd:element ref="po:class" minOccurs="1" maxOccurs="unbounded"/>
117       </xsd:sequence>
118     </xsd:complexType>
119   </xsd:element>
120
121
122   <!--
123   Some additional information at the beginning.
124   Should be overridable on the command line.
125   -->
126   <xsd:element name="head">
127     <xsd:complexType>
128       <xsd:sequence>
129         <!-- Should we create a ".jml" specification or embed the
130               annotations in existing ".java" files ? -->
131         <xsd:element name="target" type="po:ToolTarget"/>
132
133         <!-- What style of Universe annotations should we use? -->
134         <xsd:element name="style" type="po:ToolStyle"/>
135
136         <!-- Maybe the source of the annotations. -->
137         <xsd:element name="comment" type="xsd:string"/>
138       </xsd:sequence>
139     </xsd:complexType>
140   </xsd:element>
141
142
143   <!--
144   The annotations for one class .
145   -->
146   <xsd:element name="class">
147     <xsd:complexType>
148       <xsd:sequence>
149         <!-- Annotations for the fields of the class . -->
150         <xsd:element ref="po:field " minOccurs="0" maxOccurs="unbounded"/>
151
152         <!-- Annotations for the methods of the class . -->
153         <xsd:element ref="po:method" minOccurs="0" maxOccurs="unbounded"/>
154
155         <!-- Annotations for the object initializers . -->
156         <xsd:element name="object_init" type=" po:object_class_init "
157                       minOccurs="0" maxOccurs="unbounded"/>
```

```
158
159         <!-- Annotations for the class  initializers . -->
160         <xsd:element name="class_init" type=" po:object_class_init "
161                      minOccurs="0" maxOccurs="unbounded"/>
162       </xsd:sequence>
163
164       <!-- The fully qualified name of the class . -->
165       <xsd:attribute name="name" type="xsd:string" use="required"/>
166
167       <!-- Optionally, the relative path to the source file . -->
168       <xsd:attribute name="file" type="xsd:string "/>
169     </xsd:complexType>
170   </xsd:element>
171
172
173   <!--
174   The annotation for a field .
175   -->
176   <xsd:element name="field">
177     <xsd:complexType>
178       <xsd:sequence>
179         <!-- The annotations for the field initializer . -->
180         <xsd:element ref=" po:field_init " minOccurs="0" maxOccurs="1"/>
181       </xsd:sequence>
182
183       <!-- The name of the field. -->
184       <xsd:attribute name="name" type="xsd:string" use="required"/>
185
186       <!-- The Java type of the field . -->
187       <xsd:attribute name="type" type="xsd:string" use="required"/>
188
189       <!-- Optionally, the source line of the declaration .
190            Would this really help a tool to insert the annotation?
191            What if there is more than one declaration per line ?
192       -->
193       <xsd:attribute name="line" type="xsd:int "/>
194
195       <!-- One of the Universe modifiers. -->
196       <xsd:attribute name="modifier" type="po:UniverseModifier" default=" implicit_peer "/>
197     </xsd:complexType>
198   </xsd:element>
199
200
201   <!--
202   The annotations for a method or constructor .
203   -->
204   <xsd:element name="method">
205     <xsd:complexType>
206       <xsd:sequence>
207         <!-- The annotation for the return type. -->
208         <xsd:element ref="po:return" minOccurs="0" maxOccurs="1"/>
209
210         <!-- The annotations for the parameter types. -->
211         <xsd:element ref="po:parameter" minOccurs="0" maxOccurs="unbounded"/>
212
213         <!-- The annotations for the local variables . -->
214         <xsd:element ref="po:local" minOccurs="0" maxOccurs="unbounded"/>
215
```

```
216         <!-- The annotations for object creations in this method. -->
217         <xsd:element ref="po:new" minOccurs="0" maxOccurs="unbounded"/>
218
219         <!-- The annotations for casts in this method. -->
220         <xsd:element ref="po:cast" minOccurs="0" maxOccurs="unbounded"/>
221
222         <!-- The annotations for casts in this method. -->
223         <xsd:element ref="po:addcast" minOccurs="0" maxOccurs="unbounded"/>
224
225         <!-- The annotations for static calls in this method. -->
226         <xsd:element ref=" po:static_call " minOccurs="0" maxOccurs="unbounded"/>
227       </xsd:sequence>
228
229       <!-- The name of the method. Multiple methods can have the
230            same name, the parameters resolve the overloading. -->
231       <xsd:attribute name="name" type="xsd:string" use="required"/>
232
233       <!-- Optionally, the source line of the declaration.
234            Would this really help a tool to insert the annotation?
235            What if there is more than one declaration per line? 
236       -->
237       <xsd:attribute name="line" type="xsd:int"/>
238
239       <!-- Modifiers that should be added to the method.
240            At the moment there is only "pure" or "". -->
241       <xsd:attribute name="modifier" type="po:UniverseMethodModifier" default=""/>
242     </xsd:complexType>
243   </xsd:element>
244
245
246   <!--
247   The annotations for a field initialzier .
248   -->
249   <xsd:element name="field_init">
250     <xsd:complexType>
251       <xsd:sequence>
252         <!-- The annotations for object creations in this initializer . -->
253         <xsd:element ref="po:new" minOccurs="0" maxOccurs="unbounded"/>
254
255         <!-- The annotations for casts in this initializer . -->
256         <xsd:element ref="po:cast" minOccurs="0" maxOccurs="unbounded"/>
257
258         <!-- The annotations for casts in this initializer . -->
259         <xsd:element ref="po:addcast" minOccurs="0" maxOccurs="unbounded"/>
260
261         <!-- The annotations for static calls in this initializer . -->
262         <xsd:element ref=" po:static_call " minOccurs="0" maxOccurs="unbounded"/>
263       </xsd:sequence>
264     </xsd:complexType>
265   </xsd:element>
266
267
268   <!--
269   The annotations for an object or class initializer .
270   Careful: all the initializer blocks are merged into one of each kind
271   for execution.
272   So if the annotation information comes from the runtime inference tool,
273   the indices might be larger than expected from one initializer alone.
```

```
274 −−>
275 <xsd:complexType name="object_class_init">
276     <xsd:sequence>
277       <!−− The annotations for the local variables . −−>
278       <xsd:element ref="po:local" minOccurs="0" maxOccurs="unbounded"/>
279
280       <!−− The annotations for object creations in this method. −−>
281       <xsd:element ref="po:new" minOccurs="0" maxOccurs="unbounded"/>
282
283       <!−− The annotations for casts in this method. −−>
284       <xsd:element ref="po:cast" minOccurs="0" maxOccurs="unbounded"/>
285
286       <!−− The annotations for casts in this method. −−>
287       <xsd:element ref="po:addcast" minOccurs="0" maxOccurs="unbounded"/>
288
289       <!−− The annotations for static calls in this method. −−>
290       <xsd:element ref=" po:static_call " minOccurs="0" maxOccurs="unbounded"/>
291     </xsd:sequence>
292
293     <!−− The index of the initializer within the class ,
294           starting from zero .
295     −−>
296     <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
297
298     <!−− Optionally, the source line of the opening "{". −−>
299     <xsd:attribute name="line" type="xsd:int"/>
300
301     <!−− Modifiers that should be added to the method.
302           At the moment there is only "pure" or "".
303           Not supported yet, but might come...
304           <xsd:attribute name="modifier" type="UniverseMethodModifier" default=""/>
305     −−>
306 </xsd:complexType>
307
308
309 <!−−
310 The annotation for the return type.
311 −−>
312 <xsd:element name="return">
313   <xsd:complexType>
314     <!−− The Java type of the return value. −−>
315     <xsd:attribute name="type" type="xsd:string" use="required"/>
316
317     <!−− Optionally, the source line of the declaration .
318           Would this really help a tool to insert the annotation?
319           What if there is more than one declaration per line ?
320     −−>
321     <xsd:attribute name="line" type="xsd:int"/>
322
323     <!−− One of the Universe modifiers. −−>
324     <xsd:attribute name="modifier" type="po:UniverseModifier" default=" implicit_peer "/>
325   </xsd:complexType>
326 </xsd:element>
327
328
329 <!−−
330 The annotation for a parameter.
331 −−>
```

```
<xsd:element name="parameter">
  <xsd:complexType>
    <!-- The index of the parameter, starting from zero.
         Might be the only thing available. -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

    <!-- The Java type of the parameter. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>

    <!-- The name of the parameter, if available.
         Otherwise "param" + index is used as name if needed. -->
    <xsd:attribute name="name" type="xsd:string"/>

    <!-- Optionally, the source line of the declaration.
         Would this really help a tool to insert the annotation?
         What if there is more than one declaration per line?
    -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- One of the Universe modifiers. -->
    <xsd:attribute name="modifier" type="po:UniverseModifier" default="implicit_peer"/>
  </xsd:complexType>
</xsd:element>


<!--
The annotation for a local variable.
-->
<xsd:element name="local">
  <xsd:complexType>
    <!-- The index of the local variable, starting from zero.
         Might be the only thing available. -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

    <!-- The Java type of the local variable. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>

    <!-- The name of the local variable, if available.
         Otherwise "local" + index is used as name if needed. -->
    <xsd:attribute name="name" type="xsd:string"/>

    <!-- Optionally, the source line of the declaration.
         Would this really help a tool to insert the annotation?
         What if there is more than one declaration per line?
    -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- One of the Universe modifiers. -->
    <xsd:attribute name="modifier" type="po:UniverseModifier" default="implicit_peer"/>
  </xsd:complexType>
</xsd:element>


<!--
The annotation for an object creation.
The existing new expressions in a method are indexed, starting from zero.
-->
<xsd:element name="new">
```

```
390   <xsd:complexType>
391     <!-- The index of the new, starting from zero. -->
392     <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
393
394     <!-- The Java type of the new. -->
395     <xsd:attribute name="type" type="xsd:string" use="required"/>
396
397     <!-- Optionally, the source line of the new.
398          Would this really help a tool to insert the annotation?
399          What if there is more than one new per line?
400     -->
401     <xsd:attribute name="line" type="xsd:int"/>
402
403     <!-- One of the Universe modifiers. -->
404     <xsd:attribute name="modifier" type="po:UniverseModifier" default="implicit_peer"/>
405   </xsd:complexType>
406 </xsd:element>
407
408
409 <!--
410 The annotation for a cast.
411 At the moment this is very limited.
412 The existing casts in a method are indexed, starting from zero.
413 No new casts can be introduced.
414 How could we exactly say where a new cast should be inserted??
415 -->
416 <xsd:element name="cast">
417   <xsd:complexType>
418     <!-- The index of the cast, starting from zero. -->
419     <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
420
421     <!-- The Java type of the cast. -->
422     <xsd:attribute name="type" type="xsd:string" use="required"/>
423
424     <!-- Optionally, the source line of the cast.
425          Would this really help a tool to insert the annotation?
426          What if there is more than one cast per line?
427     -->
428     <xsd:attribute name="line" type="xsd:int"/>
429
430     <!-- One of the Universe modifiers. -->
431     <xsd:attribute name="modifier" type="po:UniverseModifier" default="implicit_peer"/>
432   </xsd:complexType>
433 </xsd:element>
434
435
436 <!--
437 The annotation for an additional cast.
438 At the moment we only support static type inference tools.
439 -->
440 <xsd:element name="addcast">
441   <xsd:complexType>
442
443     <!-- The Java type of the cast. -->
444     <xsd:attribute name="type" type="xsd:string" use="required"/>
445
446     <!-- Optionally, the source line of the cast.
447          Would this really help a tool to insert the annotation?
```

```
448             What if there is more than one cast per line?
449         -->
450         <xsd:attribute name="line" type="xsd:int" use="optional"/>
451
452         <!-- One of the Universe modifiers. -->
453         <xsd:attribute name="modifier" type="po:UniverseModifier" use="required"/>
454
455         <!-- One of the possible postitions to insert casts method, assignment or
456             array initializer -->
457         <xsd:attribute name="position_type" type="po:CastPositionType" use="required"/>
458
459         <!-- The index of the position. e.g the 5th method -->
460         <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
461
462         <!-- The position in the position type. -1 means target and positive
463             values means the parameter at this position.
464             In an assignment 0 is the expression to assign. -->
465         <xsd:attribute name="position" type="xsd:int" use="required"/>
466
467     </xsd:complexType>
468 </xsd:element>
469
470
471 <!--
472 The annotation for a static method call.
473 The existing static method calls in a method are indexed, starting from zero.
474 -->
475 <xsd:element name="static_call">
476     <xsd:complexType>
477         <!-- The index of the static call, starting from zero. -->
478         <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
479
480         <!-- The Java type of the call. -->
481         <xsd:attribute name="type" type="xsd:string" use="required"/>
482
483         <!-- Optionally, the source line of the call.
484             Would this really help a tool to insert the annotation?
485             What if there is more than one new per line?
486         -->
487         <xsd:attribute name="line" type="xsd:int"/>
488
489         <!-- One of the simple Universe modifiers, because static calls are
490             not possible on array types.
491         -->
492         <xsd:attribute name="modifier" type="po:SimpleUniverseModifier"
493                        default="implicit_peer"/>
494     </xsd:complexType>
495 </xsd:element>
496
497
498 </xsd:schema>
```

Appendix E

Configuration XML Schema

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="configuration">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element ref="observers" minOccurs="0" maxOccurs="1"/>
7          <xs:element ref="algorithm" minOccurs="1" maxOccurs="1"/>
8        </xs:sequence>
9      </xs:complexType>
10   </xs:element>
11
12 <!-- the list of observers that have to be instantiated by the tool -->
13   <xs:element name="observers">
14     <xs:complexType>
15       <xs:sequence>
16         <xs:element ref="observer" minOccurs="0" maxOccurs="unbounded"/>
17       </xs:sequence>
18     </xs:complexType>
19   </xs:element>
20
21 <!-- defines the classname of an observer to be instantiated -->
22   <xs:element name="observer">
23     <xs:complexType>
24       < xs:attribute name="classname" type="xs:string" use="required" />
25     </xs:complexType>
26   </xs:element>
27
28 <!-- the list of GraphVisitors that defined the algorithm and
29      have to be instantiated by the tool -->
30   <xs:element name="algorithm">
31     <xs:complexType>
32       <xs:sequence>
33         <xs:element ref=" visitor " minOccurs="1" maxOccurs="unbounded"/>
34       </xs:sequence>
35     </xs:complexType>
36   </xs:element>
37
38 <!-- defines the classname of a GraphVisitor to be instantiated -->
39   <xs:element name="visitor">
40     <xs:complexType>
41       < xs:attribute name="classname" type="xs:string" use="required" />
42       <xs:sequence>
```

```
43          <xs:element ref="option" minOccurs="0" maxOccurs="unbounded"/>
44        </xs:sequence>
45      </xs:complexType>
46    </xs:element>
47
48  <!-- defines an option for a visitor with an optionname and a value
49          each visitor may have an unlimited number of options. See the sample
50          config.xml file .-->
51    <xs:element name="option">
52      <xs:complexType>
53        <xs:attribute name="name" type="xs:string" use="required" />
54        <xs:attribute name="value" type="xs:string" use="required" />
55      </xs:complexType>
56    </xs:element>
57
58  </xs:schema>
```