Runtime Universe Type Inference

Frank Lyner

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

January 12, 2005 — July 11, 2005

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 better control object aliasing and dependencies between different program parts. Java programs can be annotated with Universe types to restrict the access between different components. We developed an algorithm that infers these type annotations from executable Java programs. This is done through runtime observation and not through static code analysis. As a proof of concept the algorithm was implemented and successfully applied to real Java programs. This report covers every aspect of the information gathering and processing needed to perform this task.

Contents

Chapter 1

Introduction

1.1 Overview

In object-oriented programs, every object is allowed to reference any other object. The access to objects is provided through reference passing. This introduces dependencies between the objects which makes it difficult to reason about the correctness of programs.

Ownership provides means to structuring the object store and restricting the referencing between different objects. The Universe type system [1] enables the programmer to express these ownership rules and to check them statically without introducing a large overhead.

The goal of this project is to infer the Universe types of a given program by observing its execution and analysing its object store.

1.2 Universe Type System

1.2.1 Aliasing

Aliasing occurs, when two or more variables hold a reference to the same object. In object-oriented programming languages like Sun's Java every access to an object is done through a reference. Every object which occurs as parameter in a method call is passed by reference and not by value. This prevents the expensive copying of data and is the main reason why object-oriented programs are efficient.

On the other hand a lot of aliases are introduced. Each of these aliases enables the holder to access and modify the state of the referenced object [2]. This makes it very difficult to reason about the correctness of programs and to ensure that no consistency rules are violated. Of course, the access may be restricted by setting the visibility of the classes, methods and fields. But there is no way one can ensure, that, for instance, subclasses will adhere to these restrictions.

Problems with aliasing

Reference leaking

Listing 1.1: Reference Leaking

```
class BankCustomer {
        protected Account account;
         ...
}

class UnsafeCustomer extends BankCustomer {
        public Account getAccount() {
                return account;
        }
}
```

As shown in listing 1.1 reference leaking occurs, when a reference to a representation object of a component is passed to clients of that component. The problem is now, that a client of a UnsafeCustomer object can manipulate the account information without having to use the interface methods of that class. This is also known as *representation exposure*.

The only conventional way to prevent the UnsafeCustomer to leak the account reference would be to declare the field account as private and the class as **final**. But this would prevent any subclassing, which is too restrictive in many situations.

Reference capturing

Listing 1.2: Reference capturing.

```
class Item {
        int priority ;
}

class PriorityQueue {
        ...
        public void add(Item item) { // checks omitted
                items[last++] = item;
        }
}

void clientMethod() {
        Item item = new Item();
        item.priority = 10;
        priorityQueue.add(item);
        priorityQueue.sort();
        item.priority = 20;
}
```

A problem similar to reference leaking is reference capturing. As can be seen in listing 1.2 the priority queue includes the reference to an item into its representation. That allows a client to change the priority of an item even after the queue was sorted. This represents a violation of the invariant of the priority queue.

Reference leaking and reference capturing are similar since the result of both is the same. In both cases a client module gains access to the representation of the structure under consideration and can therefore change its state by bypassing its interface. This results in violations of the invariants of the structure.

1.2.2 Ownership

Several propositions where made to provide better alias control [2, 3, 4]. The one developed by Clarke, Potter and Noble is called the *Ownership Types* [5]. They introduce the notion of *object contexts*. A context is the set of objects which have the same owner. Each object owns a context and lies itself in a context that is owned by at most one object. This defines a hierarchical structure of the object store. This hierarchy is rooted at the *root context* which is not owned by a particular object. It contains all objects that are created from the main method or which have no specified owner.

The type system uses type modifiers to express the context of the referenced objects. Consider a field f of type T of a class C. The type of the field can be modified to **owner** T. That means that the objects referenced by f have the same owner as an instance of C. Therefore, they will be in the same context. if f is of type **rep** T, that means that the objects referenced by f are in the context owned by instances of the class C.

Aliasing is controlled by only allowing objects to reference other objects that are in contexts accessible to them. The accessible contexts of an object are usually the one to which the object belongs or the one that is owned by the object. Additionally, an object may gain the allowance to access contexts that are in a higher context hierarchy through context parameters.

It is a unique feature of the Ownership types that classes can be parameterised. Instances of a parameterised class are allowed to access objects in the context given as parameters. This allows a controlled access from an inner context to an outer context.

Any other crossing of context boundaries is forbidden. The direct consequence is that all reference paths that end at a given object and start outside its context, have to pass through its owner. This property is known as *ownership invariant* or *owner-as-dominator*.

1.2.3 Details

The rule to prevent any access between contexts, except to ones that where declared with parameters, is very restrictive. It does not allow to implement widely used programming techniques like iterators. Iterators are used by the client of a datastructure and are therefore outside the context of the structure. But they need access to the representation of the datastructure to be able to iterate over the elements. This is not possible with the ownership types of Clarke et al.

The representation exposure becomes only a problem, when the leaked reference is used to modify the representation. No invariant can be violated, if a reference is only used to read the state of a given object.

For this reason the Universe type system introduces the notion of **readonly** references. The typesystem ensures that **readonly** references are never used for write accesses. Because of this, they are allowed to cross any context boundary. But for write references the rule still applies that they cannot cross context boundaries except from an owner to an owned object. The owner-as-dominator property is therefore changed to the *owner-as-modifier* property. Any path of *write* reference that end at a given object and starts outside its context has to pass through its owner.

At this point the Universe type system is more restrictive the the ownership types. It does not support type parameterisation. The consequence is that controlled write references to higher contexts are not possible. These crossings of context boundaries where found to be a problem for the modular verification of program parts[6] and are not supported for this reason. Another advantage is that the syntax is simpler.

Type modifiers

To express ownership relations, type modifiers are used. When a reference should only be allowed to be used for read access, the type of the variable in which the reference is stored is annotated with the **readonly** type modifier. If a variable should hold references to objects inside the representation of its object, then its type is annotated with the **rep** type modifier. If the references should hold references to objects in the same context, the variable type is annotated with **peer**. This follows the Universe type system terminology, where objects in the same context are called *peer* of each other.

Listing 1.3: Linked list with type modifiers.

```
class Node {
        /*@ peer @*/ Node next;
        /*@ readonly @*/ Object item;
}

class List {
        /*@ rep @*/ Node first;
        /*@ rep @*/ Node last;
        int size;
}
```

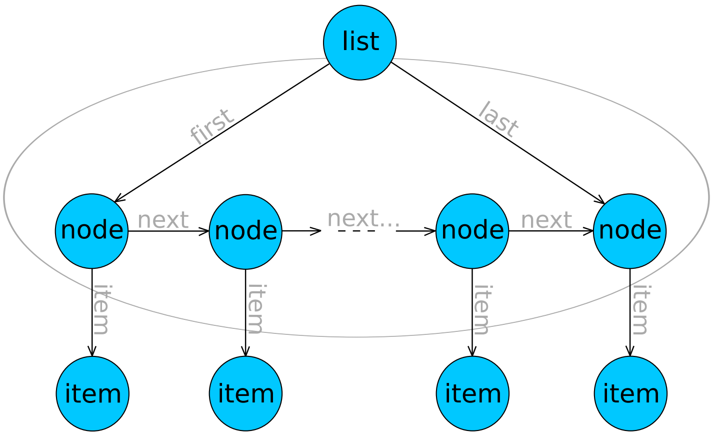

Figure 1.1: The objects of the linked list in its context.

Listing 1.3[1] and figure 1.1 show the ownership relations of a linked list and how they are declared using the Universe type system. Of course the owner of the list context is an object of type List. The objects of type Node belong to its representation. That is why the variables first and last, which hold references to node objects, are annotated with **rep**. All node objects belong to the representation of the list and therefore to its context. Because they are peer to each other, the variable next is annotated with **peer**. The items are passed to the list to be stored. They are created in arbitrary contexts. To be able to store references that cross the context boundaries, the variable item is annotated with **readonly**. This imposes no restrictions as the list should never modify the items it stores. With the type modifier **readonly** it is ensured that this never will happen.

Method calls and pure methods

When an object calls a method upon another object, the state of the target object or of objects referenced by it, might be changed. For this reason a method call is considered a write access on the target object. This is forbidden for readonly references.

To forbid any method call on a readonly reference is too restrictive. For instance, the Object.equals() method in Java does not change anything and should be allowed to be called on a readonly reference.

This is done by introducing the notion of pure methods. A pure method is explicitly marked with the keyword **pure**. It is not allowed to perform write operations on any objects and can only call other pure methods and constructors. The references passed as parameters can not be used to perform write operations and have to be of type **readonly**.

Type rules

As stated above the Universe type system defines type modifiers for reference types. Given the class T, the type **rep** T is not equal the type **peer** T.

Subtyping

The subtype relation is as in Java. Given the type T and its subtype S, denoted by $T > S$, the following rules hold:

$$\begin{aligned} peer\ T &> peer\ S \\ rep\ T &> rep\ S \end{aligned}$$

[1] The notation used in this listing is the one defined by JML [7], which was adapted to support the Universe type system[8].

$$readonly\ T \quad > \quad readonly\ S$$

Additionally, there is a subtype relation between the type modifiers:

$$readonly\ T > peer\ T$$
$$readonly\ T > rep\ T$$

Intuitively, this says that readonly does not state anything about the referenced context. In particular, objects in the same context or that are owned by the current object, can also be referenced by a readonly reference.

Assignment

The type rules for assignment are as in Java. The left hand side of an assignment has to be the same or a supertype of the right hand side.

Field access

To determine the type of a field access expression `x.f` both type modifiers of x and f have to be considered (as defined by and taken from [8]):

1. If the types of both x and f are peer types, then we know (a) that the object referenced by x has the same owner as **this**, and (b) that the object referenced by x.f has the same owner as x and, thus, the same owner as this. Consequently, the type of x.f also has the modifier **peer**.

2. If the type of f is a rep type, then the type of **this**`.f` has the modifier **rep**, because the object referenced by **this**`.f` is owned by **this**.

3. If the type of x is a rep type and the type of f is a peer type, then the type of x.f has the modifier **rep**, because (a) the object referenced by x is owned by **this**, and (b) the object referenced by x.f has the same owner as x, that is, **this**.

4. In all other cases, we cannot determine statically that the object referenced by x.f has the same owner as this or is owned by this. Therefore, in these cases the type of x.f has the modifier **readonly**.

Method parameters

For method calls the types of the actual parameters have to be checked against the types of the formal parameters. This is done w.r.t the target of the method call. That means, that if a formal parameter has the type modifier **peer** then the referenced object has to be peer to the target of the call. Therefore, if the caller invokes the method on an **rep** reference, the actual parameter passed has to be **rep**, since it has to be peer to the target object. The same rules apply as for field accesses. The method return type can be seen as just another parameter and is therefore handled the same way.

Arrays

Array types need two type modifiers to declare their contexts. The first one declares the context of the array object itself. The second one is needed to declare the context of its elements. The type /*@ *rep peer* @*/T[] is the type of an array owned by **this** which elements are **peer** to the array object, and therefore also in the context owned by **this**. In other words, the second type modifier is to be interpreted relative to the context of the array object.

The second type modifier is only needed when the array fields are of reference type. Arrays of primitive types can be annotated like normal fields with only one type modifier. For the rest of this section, arrays are assumed to be of reference type.

Array objects are only used to access other objects. They do never perform an operation on an object referenced by one of their fields. This is the reason why the second type modifier can never be **rep**.

Multidimensional arrays are also annotated with two type modifiers. Conceptually, they are not different from one dimensional arrays apart from additional dimensions. But in Java they are implemented by one dimensional arrays that store references to other arrays. Because these references cannot be annotated, all array objects that represent a multidimensional array have to be in the same context.

1.3 Goal

The goal of this project is to develop an algorithm that annotates a regular Java program with Universe types. This is done dynamically, through observation of program execution and not statically, through source code analysis. This is also known as dynamic type inference.

The algorithm has to be implemented to provide a proof of concept.

1.4 Project Scope

The implemented algorithm should be able to gather all necessary information through program tracing, to infer the Universe types and to print out the found annotations. The incorporation of the annotation into the Java code under consideration is not part of this project.

Furthermore, some restrictions about the supported language structures are done. Following is a list of features that do not belong to the project scope:

Static Methods/ Static Fields Static methods and fields add a lot of complexity to the problem, since they do not belong to a specific object. In the case of static fields, it is not yet fully determined, how they are handled in the Universe type system.

Static methods have to be handled by the program to be developed, as every Java program has one static `main()` method. But this can be done in a simplified way as will be discussed in section 2.5.1 on page 33.

Arrays As will be described later, it is very difficult to gather information about the access or modification of array fields (see section 1.2.3 and section 4.3.1).

Multithreading For simplicity's sake, it was decided to restrict the scope of the project to programs with only one thread.

The focus is set on the annotation of method and field signatures. To annotate local variables is not part of the requirements of this project.

As described above, the information is gathered through observation of program execution. This raises immediately the question about the choice of input values and code coverage. This is a very complex problem which is very hard to solve automatically. It was decided that it is the responsibility of the user of the implemented tool, to provide input values that will lead to the desired code coverage.

1.5 Outlook

First the developed algorithm is described in chapter 2. Then we will discuss some implementation details in chapter 3. This chapter serves also as documentation of the developed type inferring tool. In the end we give some information about time and memory consumption and discuss the results of this project.

Chapter 2

Algorithm

In this chapter the Universe type inferring algorithm is introduced. The basic idea is to identify the owner of each object created by the program during a sample execution and to then map this dynamic structure to the static structure of the corresponding classes. By inferring ownership on each object the contexts are determined. This information will then be needed to determine the type of a given field or method parameter.

First we will introduce the representation chosen to reason about ownership of objects. Then the algorithm will be introduced. The basic approach of inferring ownership is to first compute an approximation which will set a basic layout but also introduce some conflicts. Then these conflicts will be resolved and the contexts will be defined. At the end the annotations found for each object will be harmonised to be able to map them to their corresponding class.

First, the algorithm is described in a somewhat abstract manner. Then, it will be explained step by step in detail in section 2.5.

2.1 Overview of the steps of the algorithm

1. Information Gathering

   (a) Build datastructure (EOG) through monitoring the program execution (section 2.5.1).

2. Structuring the Object Store in Universes (abstract description in section 2.3)

   (b) Build dominator tree upon EOG (section 2.5.2).

   (c) Store depth in dominator tree in each node (section 2.5.3).

   (d) Determine priority for each edge and build worklist (section 2.5.4).

(e) Resolve conflicts through processing of the worklist (section 2.5.5).

3. Finding Valid Annotations (abstract description in section 2.4)

(f) Harmonise and annotate (section 2.5.6).

(g) Generate output (section 2.5.7).

2.2 Representation

2.2.1 Object Graph

As object-oriented programs are executed, objects are allocated on the heap. They receive references to other objects that allow them to inspect and modify their state. Eventually they will be deallocated. To be able to reason about the object structure we need a representation. It is common to use the Object Graph. Each object in the heap is represented as a node in the Graph. The references between the objects are mapped to the directed edges of the graph.

2.2.2 Extended Object Graph

An Object Graph is a snapshot of the heap at a certain point in time. The changes over time can not be captured. To get a complete representation of the heap during the whole execution time, one could draw an Object Graph after each state change. However, this would lead to a large number of graphs with a lot redundant information. The state changes, like the addition of a reference, tend to yield small changes in the object graph.

In the context of ownership inference it is not necessary to represent all changes that occur over the execution time. When an object A performs a write operation upon another object B, then we know that A has to be the owner of B or—in the Universe type system—is a peer of B. It does not matter if A or B or both will be deleted or if A looses the reference to B later in the execution. In [9] an alternative representation is formally introduced for exactly that purpose. It is called the Extended Object Graph.

The Extended Object Graph can be seen as the overlay of all object graphs of a program execution. Each object ever created by the program is represented. Each reference that was present at a given point in time will also be present. In other words, only the addition of objects or references to the graph is recorded, but not the deletion. This is illustrated in figure 2.1.

In the Universe type system a distinction is made between readonly references and write references. For the goal of determining the owner of each object, the readonly references

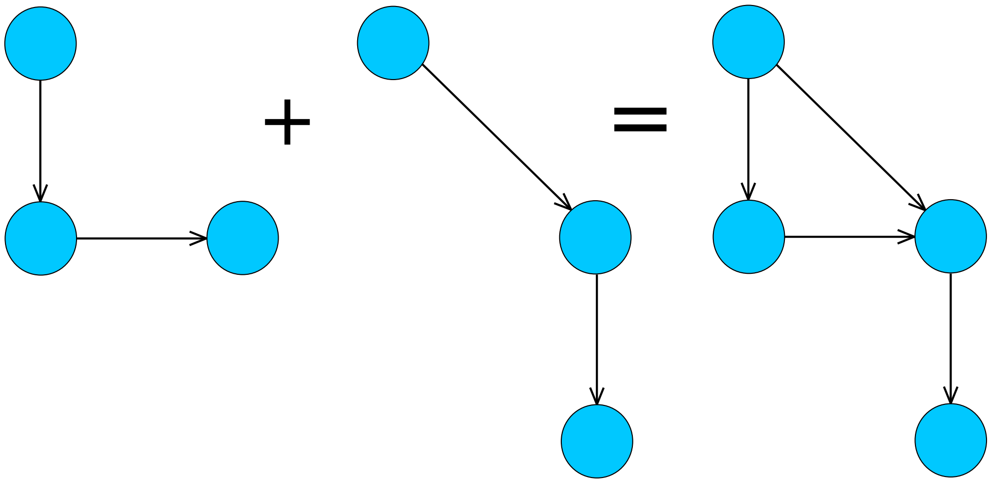

Figure 2.1: The Extended Object Graph is the overlay of all object graphs of a program execution.

are not necessary. Each object may have a readonly reference to any other object without violating the Universe type rules. Therefore, we will omit these for the moment for the discussion of how the ownership is inferred upon the Extended Object Graph. So when we will mention the Extended Object Graph, it will contain only the write references.

Later we will need to record in which object field a reference was stored. Then we will need to track the readonly references, too.

2.3 Inferring Ownership

We assume that the Extended Object Graph has been completely built up and that it contains all write references. The goal is to assign to each object its owner. This will create a context hierarchy which should preferably be as deep as possible. Every program can quickly be annotated with Universe types by declaring all fields and method parameters and return values as **peer**. But this would be of no use whatsoever. On the other hand, an annotation that yields a deep context hierarchy will provide a good structuring of the objects and will help to better control dependencies and aliasing.

The Universe type system has a notion of a root context. For instance all objects created by the `main()` method belong to the root context. We want to assign the owner to each object, therefore we need an owner of the root context. For this, we introduce a root object that has no correspondence to any real object.

2.3.1 Owner is approximated by direct dominator

When an object A creates an object B by using the `new` operator we introduce a write edge from A to B. If B is created by the `main()` method we introduce a write edge form the root to B. Therefore, with the root object the Extended Object Graph is connected.

Only the owner or peers of an object are allowed to write it. Therefore, every path over the write edges in the Extended Object Graph from the root to an object O goes through the owner of O. Please note, that this corresponds exactly to the definition of *dominators* for flowgraphs used by compilers. The Extended Object Graph has exactly the same definition as the flowgraph; a directed graph with a root. Therefore, it is valid to introduce the notion of dominators for the Extended Object Graph. A *direct dominator* of a node N is a dominator of N that is dominated by all other dominators of N. In other words, when following a path from the root to N, the last node that fullfills the dominator requirement is the direct dominator of N.

The owner of an object has to be one of its dominators. This property is also known as *owners as dominators* [10, 11, 5]. It provides us with a very good approximation. If we take the direct dominator as approximation for the owner, this yields the deepest possible hierarchy. Unfortunately, some conflicts will be introduced.

The problem of finding dominators has been widely researched in the context of compiler optimisation. A fast algorithm was developed by Lengauer/Tarjan [12]. Several other algorithms are known [13, 14], but the Lengauer/Tarjan is the most widely used and can be implemented very efficiently.

2.3.2 Possible conflicts after approximation

With the approximation, the only conflicts possible are write references from a deeper context to a higher context. If there would be conflicting edges from a higher context to a deeper one, the dominator algorithm would have produced an other result.

In figure 2.2 the contexts supposedly found by the dominator algorithm are represented by the gray ellipses. So object A is the dominator of B and B is the dominator of D. The edge C—D crosses the boundary of the context owned by B, which is illegal in the Universe type system and is therefore considered to be a conflict. Clearly, the edge C—D represents another path from the root to the object D. Therefore, the dominator algorithm will not assign B as the dominator of D but A. This makes C and D to peers and the conflict can not happen.

Here are three cases which will yield conflicting edges:

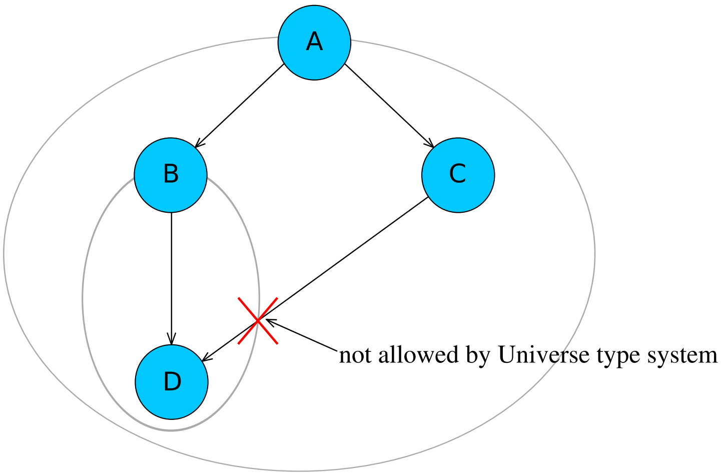

Figure 2.2: A hypothetical scenario, where a conflicting edge connects a higher context with a lower one.

Cycle

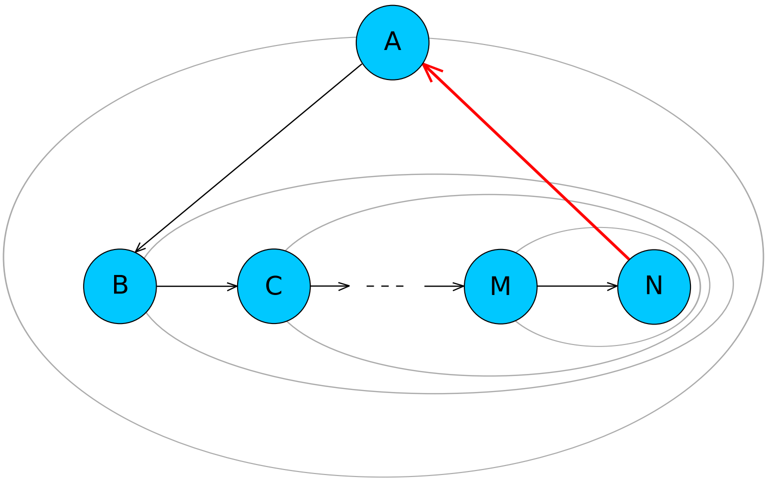

Figure 2.3: Edge N—A is a conflicting edge since it crosses context boundaries, that where determined by the dominator algorithm.

In the presence of a cycle like shown in figure 2.3, the dominator algorithm will assign A as the dominator of B, B as the dominator of C and so on until object N. The edge N—A will then be a conflicting edge, because it crosses context boundaries. Of course N will have to be made peer to A because it has a write edge to it. It cannot be the owner of A because it is not a dominator of A. But by raising N to the same context as A the edge M—N will become a conflicting edge. This shows that all objects B, ..., N will have to be made peer to A.

False cycle I

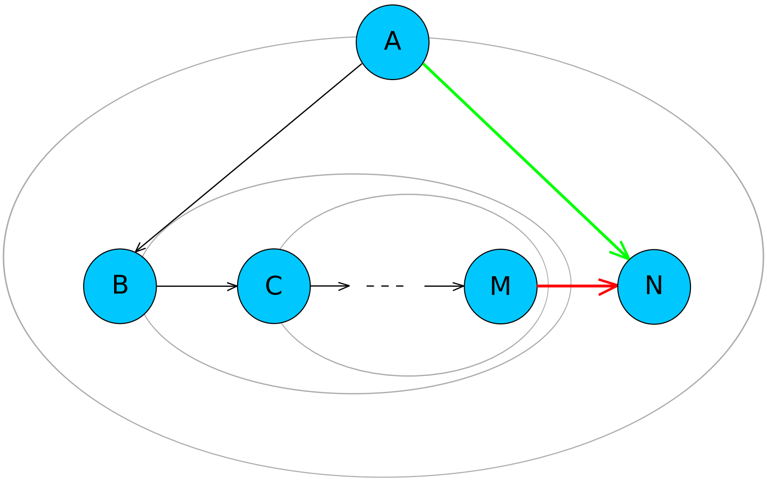

Figure 2.4: The edge A—N is not a conflicting edge, but the edge M—N.

As can be seen in figure 2.4 a false cycle is a cycle with one edge pointing in the opposite direction (here edge A—N). Here the edge A—N is not a conflicting edge and A will be found as the direct dominator of N. The problem arises at the edge M—N. N is in the context owned by A but M is in a much deeper context. The solution is quite similar to the one above. M will have to be made peer to N, which will introduce a conflict which will have to be resolved in the same way. The difference to the cycle problem is that A will be the owner of all object under consideration.

False cycle II

Conflicting edges can also occur where there is no edge between an object and its owner. In the example in figure 2.5, N is in the context of A since A is the last object in which the two paths from the root to N are joined. Of course the edges M—N and O—N represent conflicting edges. Again all objects under A will have to be raised to the context owned by A, thus becoming peer to each other.

If an object A is raised to a higher context, no new conflicts can be introduced with objects that are in the context owned by A. If there where no conflicts before, then there is no write edge crossing the context boundary in either direction. If this where not true, either A would not be the owner of that context or the outgoing reference would represent a conflict on its own. This property is very important as it shows that changing the owner of an object does not affect the objects in its context or in one of its subcontext.

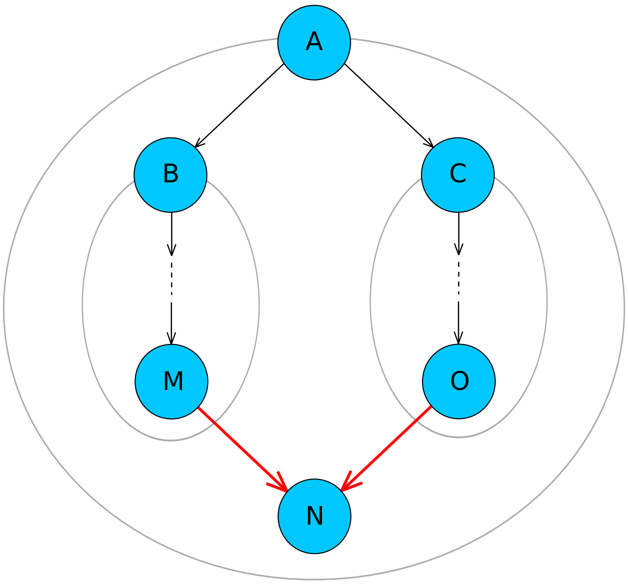

Figure 2.5: N is in the context of A, making the edges M—N and O—N to conflicts.

2.3.3 Conflict resolution

All conflict resolutions described above involved identifying objects that need to be made peer. This is quite a difficult task because of all the interdependencies. Every object that is raised to another context may have other references that may become new conflicts. This in turn may require a reevaluation of the already raised objects. Apart from efficiency considerations, an algorithm has to be found that guarantees to terminate.

The discussion of the different conflicts possible already implied a resolution approach: Find a conflicting edge in the Extended Object Graph, set the owner of the start object to the owner of the end object and follow in- and out-going edges recursively that introduced new conflicts.

In figure 2.6, the edge D—O represents a conflict. To resolve it D has to be made peer to O. This is done by setting the owner of D to the owner of O. Now, the edge C—D has become a conflict. Again, the owner of C is set to the owner of D. The same is repeated for the edges B—C and A—B. The algorithm will then also check the edge O—A but it will find that O and A are already peer to each other, so nothing has to be done. Of course also edge B—E is checked but as described in 2.3.2, no new conflicts can be introduced from the context owned by the raised object.

The algorithm relies on the fact that after the approximation has been computed, the end object of a conflicting edge will always be in a higher context than the start object. This was already explained at the beginning of section 2.3.2.

This explains, why the owner of the start object has to be set to the owner of the

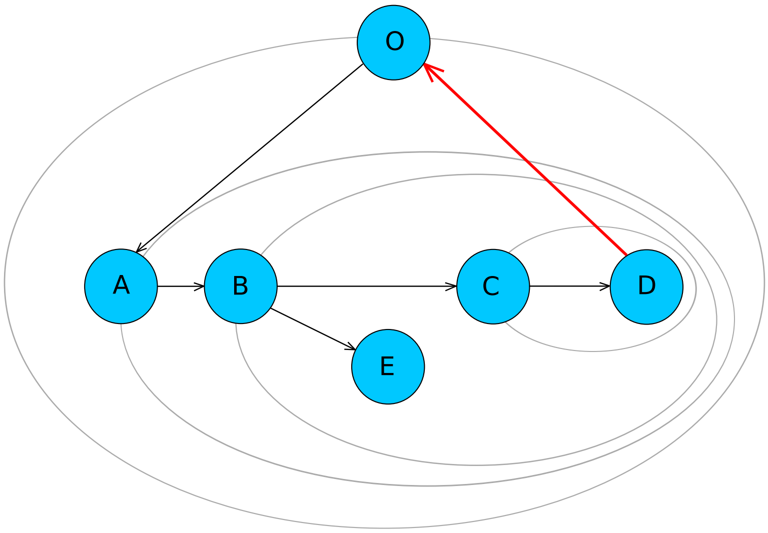

Figure 2.6: Objects O, A, B, C and D will be made peer. Object E stays in the context of B.

end object and not the other way around. In order to process all conflicting edges, the algorithm builds up a worklist of these edges. This worklist is then processed, one conflict at a time, until it is empty.

It would be most efficient and termination could be ensured if it could be shown that each edge is only processed at most once. For this several problems have to be solved. But we will show that all of them can be solved by processing the edges in a given order.

Top-down

The edges have to be processed in a top-down manner with respect to the context level. Consider the example shown in figure 2.7.

The result of the dominator algorithm would be that A is the owner of B and B is the owner of C. The edges C—B and B—A are the conflicting edges. If the algorithm would begin by resolving C—B first, then C would be made peer to B. The edges A—B and B—A do not need to be followed in this step as they represent no new conflicts. In a second step the conflict B—A would be processed and B would be made peer to A. Now, the edge C—B has again become a conflict and the edges B—C and C—B would have to be processed again to resolve it. Clearly, the goal is not achieved.

If the algorithm would start at the edge B—A, it could make B peer to A and later C peer to B. No edges would have to be processed twice.

From this observation, we conclude, that the algorithm must start to resolve those conflicts that appear highest in the context hierarchy and move down accordingly.

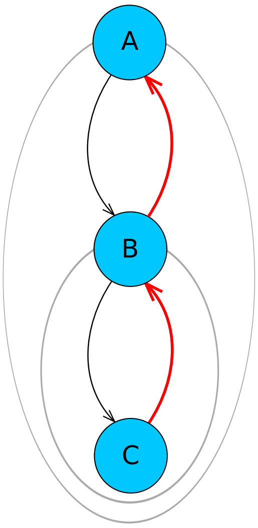

Figure 2.7: If the conflicting edge C—B is resolved before B—A, then the edges B—C and C—B will have to be revisited.

Nested cycles

Another problem appears, when the following situation arises:

If two cycles are nested, the order in which the corresponding conflicts are resolved is relevant. In the example shown in figure 2.8, if the conflicting edge C—B is processed first, then C is made peer to B. Then the edge D—A would be processed and—during the recursive conflict resolution—the edges B—C and C—D would be processed again. If on the other hand, the ”big” cycle is processed first, then B and C would be made peer in this step already. The conflict C—B would not exist anymore. Please note that this is only true if the ”big” cycle is completely processed, before the ”small” cycle is resolved.

But how do we distinguish ”big” from ”small” cycles? Intuitively it is simple, the cycle with more nodes is bigger then the one with less nodes. In the context of the Universe type system it is not the amount of nodes that count, but the amount of contexts that a cycle crosses. For each context covered by the cycle, one conflicting edge will be introduced and will have to be followed, when the corresponding node is raised. By doing this, all smaller cycles that are present on a subset of the nodes of the big cycle, will be resolved.

As stated above, all conflicting edges are part of a cycle, either a true or a false one. It is obvious that a conflicting edge will always connect the node in the deepest context with the node in the highest context. Therefore, the size of a cycle can immediately be determined by comparing the context levels of the start and the end object of the conflicting edge. Since we approximate the owner with the director dominator, the context level corresponds directly to the height of an object in the dominator tree. Because the direct dominators are only an approximation, we prefer to use the term dominator level

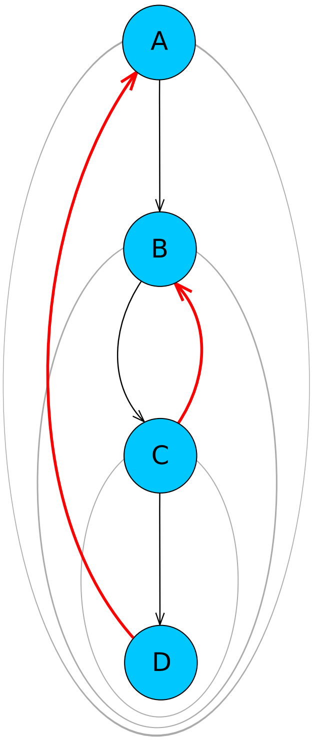

Figure 2.8: If the conflicting edge C—B is resolved before D—A, then the edges B—C and C—B will have to be revisited.

rather than context level.

Conclusion

These two rules—top-down and big-before-small—lead to the definition of a priority with which a conflicting edge should be processed. The dominator level starts at the root where it is 0 and increases then with the height in the dominator tree. For each edge the smallest dominator level of its two nodes is determined. We will call it the minimum dominator level. Additionally, the dominator level difference between the two nodes is computed. An edge has a higher priority than another if its minimum dominator level is smaller then the one of the other edge. If two edges have the same minimum dominator level, the edge with the higher dominator level difference is chosen.

Mathematically, this can be formulated as follows:

The priority of an edge $A \rightarrow B$ is a tuple $(min\_dom\_lvl, dom\_lvl\_diff)$, where

$$min\_dom\_lvl = min(A.dominatorLevel, B.dominatorLevel)$$

and

$$dom\_lvl\_diff = abs(A.dominatorLevel - B.dominatorLevel)$$

Given two edges E_1 and E_2, E_1 has higher priority if:

$$E_1.min\_dom\_lvl < E_2.min\_dom\_lvl$$

or if equal:

$$E_1.dom\_lvl\_diff >= E_2.dom\_lvl\_diff$$

There are two places where the priority will take effect.

1. *Sorting the Worklist:* The worklist of conflicting edges is sorted according to the priority. This ensures, that the conflicts are resolved in a top down manner and that conflicts of large cycles are resolved before conflicts of small cycles.

2. *Sort edges to follow recursively:* For each object that has been raised, all in- and outgoing references are followed. These references are sorted according to their priority and followed in that order. In that way, the edge that connects an object with its owner will always be processed first. This ensures, that the large cycles are completely processed, before the resolution of the small cycles begins.

2.4 Harmonisation and Annotation

The final goal of this algorithm is to annotate the field and method signatures with Universe types. That means that there are three entities which types need to be inferred: fields, method parameters and method return values. The term we will use to refer to one of these entities will be *variable*.

To be able to determine the type of a variable we need to know which references where stored in this variable and in what context the referenced objects are. Ideally we would like to know which variable was used to perform a given write operation. Here is an example:

Listing 2.1: write operation

```
void setFoo(Value newFoo) {
        foo = newFoo;
        foo.writeSomething();
}
```

We see that the variable newFoo is never used to perform a write operation. Instead, the reference stored in newFoo is stored to the field foo, which is then use to call a non-pure method. The correct annotation would be **readonly** for newFoo and **rep** or **peer** for foo.

The problem is that, due to limitations in the tracing capabilities, it can not be determined that foo was used to perform the method call. What we can find out is that the references that are passed to this method as actual parameters—stored in newFoo—, are stored in the field foo. We know this, because this raises a field modification event.

In other words, we are able to find out which reference was stored into which variable, but are not able to discover how this reference was used. This knowledge about which reference was stored in which variable needs to be inserted into the graph.

2.4.1 Adapting the Extended Object Graph

As stated above, we are not able to figure out which variable was used to perform a write operation. Therefore, a write reference that is not related to any variable is the best we can do. On the other hand, when a reference is written into a variable, the variable can be identified. That is why a new type of edges is added to the Extended Object Graph that is called a *variable reference*. A variable reference is also a directed edge and has a variable attached to it. It is inserted into the graph if at any time, during the execution of the program, the start object held a reference to the end object stored into the corresponding variable.

The picture 2.9 on page 35 shows a part of the Extended Object Graph after the field modification operation in listing 2.2. This illustrates how variable references are added to the Extended Object Graph.

2.4.2 Harmonisation

Several variable references can be inserted for the same variable. An object may assign different references to the same field, for instance. Additionally, there may be two different objects of the same class. Of course, the same variable will store different references in each object. The situation is simple when all references of the same variable are pointing into the same ”kind” of contexts.

For instance, if all references are referencing objects that are peer to the particular object, the variable could simply be annotated as **peer**. If the particular object containing the variable is the owner of all referenced objects, then a **rep** annotation would be safe to choose.

Of course, it can happen that one variable reference points to a peer and another points to an owned object. This is the case where the two references need to be harmonised. The exact mechanism will be described later in section 2.5.6 but the idea is to raise the owned object in the same context as the start object of the corresponding reference.

2.5 Detailed Description of the Algorithm

Following is the detailed description of each step of the algorithm.

2.5.1 Build datastructure (EOG)

To begin, the runtime information has to be gathered. Based on this information the Extended Object Graph is constructed. The Java Virtual Machine provides several interfaces at different abstraction levels that provide runtime information of a program. Additionally, there are a variety of third party tracing tools. The technology chosen for the project will be discussed in section 3.1. However, all these technologies have in common that they are event based and allow to inspect certain aspects of the state of the Java Virtual Machine. The mechanism is that an agent selects a number of events it is interested in and registers itself to the Java Virtual Machine to be notified when such events occur. When the agent is notified it typically receives a handle that allows it to further inspect the Java Virtual Machine.

Following is a list of events that are interesting to this project. It is explained how they are handled and what information is retrieved from each event.

Method entry

When a method entry event occurs the following information is received or additionally retrieved from the Java Virtual Machine:

- The calling object.
- The target object of the call.
- The value of each method parameter of reference type.
- The method name.
- The class that declares the method.

Here is the explanation, how the Extended Object Graph is built up:

General method entry

Generally, a method call of an object A on an object B is considered as a write operation from A on B. Normally, such a method will change the state of the target object. That is why for each method entry event a write reference is inserted in the Extended Object Graph starting at the caller and ending at the target object.

An exception to this rule is a call of a **pure** method.

Pure methods

As described above, methods declared as **pure** are not allowed to modify the state of the target object. Therefore, the caller is not considered as performing a write operation on the target and the corresponding write reference is not inserted in the Extended Object Graph.

The distinction if a method is **pure** or not is beyond the scope of this project. The list of pure methods is provided by the user at the beginning of the process.

Method parameters

The parameters are variables of the class declaring the method. When a method is called on an object and references are passed as arguments of the method, then it is known that this object will have these references. This is why variable references are inserted for each parameter of reference type. These variable references will start at the target object and will end at the object referenced by the corresponding parameter.

Object creation

When a new object is created the constructor of its class is called. This results in a method entry event of a method called `<init>`.

In this case a new object is inserted in the Extended Object Graph that corresponds to the target of the method call. Additionally, a write reference is inserted that starts at the caller and ends at the target. Thus, the creation of a new object is considered to be a write operation upon it. This is in accordance to the Universe type systemas the **new** operation can only yield **peer** or **rep** references.

The caller is generally the **this** object of the one below-top method frame on the frame stack. The target is identified as the this object of the method being entered. The problem is that in static methods the **this** object is not defined. They need special handling.

Static methods

As stated above, the correct handling of static methods is not part of the scope of this project. However, in each program at least one static method will occur, namely the main method. That is why it was considered important to allow static methods and to handle them in a simplified way.

For the main method the solution is quite simple. Since this method is—per definition—running inside the root context, the root object is taken as the **this** object. This is not only simple, but also compliant to the Universe type system rules. For simplicity's sake, it was decided that all static methods should be handled that way. This does not introduce problems as long as no static method occurs in the program under consideration, that creates new objects or that modifies its parameter objects.

Start of event handling

When the Java Virtual Machine starts up a lot of objects are created, several threads are loading missing classes and the virtual machine performs a number of other initialisation tasks. All this activity will already start to generate events, that will be the same for every start-up and are unrelated to the program under consideration. The same is also true for the end of the program execution, when the Java Virtual Machine is performing general clean up and resource releasing operations.

The solution is to ignore all incoming events until the first part of the program under consideration is executed. There are two possibilities for this code. The first is a method call to the class initialiser. If static variables are directly initialised or if a **static** {...} block is present in a class, the corresponding code will be executed in a method called <clinit>.

In such a case the first code of program executed will be the one inside the <clinit> method of the class containing the main method. In this case the program tracing will begin when a method entry event of the <clinit> method of the main class is registered.

If no static initialisation is done on the class containing the main method, the <clinit> method will not exist. In this case the program tracing will begin when the method entry event of the main method is registered.

In both cases the end of the main method indicates the end of the program. This is registered by the method exit event of the main method. From this point on, all subsequent events will again be ignored.

Method exit

When a method exit event occurs the following information is received or additionally retrieved from the Java Virtual Machine:

- The target object of the call.
- If the return value of the method is of a reference type, the referenced object.
- The method name.
- The class that declares the method.

If the method does not have a return value of reference type, the Extended Object Graph is not modified when a method exit event occurs. Otherwise, the return value is considered to be a variable of the class declaring the method. Conceptionally, the return value can be viewed as just another parameter. Therefore, like for parameters, a variable reference is inserted between the target object and the object referenced by the return value.

As stated above, another use of the method exit event is to register the end of the `main` method. When this happens the program is considered to have reached the end of its execution and all following events will be ignored.

Field modification

When a field modification event occurs the following information is received or additionally retrieved from the Java Virtual Machine:

- The object that contains that field; aka the changed object.
- The object performing the write operation.
- The object whose reference is being stored in the field, if the field is of reference type.
- The class that declares the field.

A field modification results in a state change of the object containing that field. Therefore it is interpreted as a write operation. That is why for every field modification event a write edge is inserted, starting at the object performing the write operation and ending at the object whose field is being modified.

When the field is of reference type, a field modification means that a new reference is assigned to that field. For this, a variable reference is inserted, starting at the modified object and ending at the object being referenced. The variable stored with the variable reference will be the field under modification.

Figure 2.9 illustrates the edges that are inserted for the field modification event from listing 2.2:

Listing 2.2: Field Modification

```
var.f = ref;
```

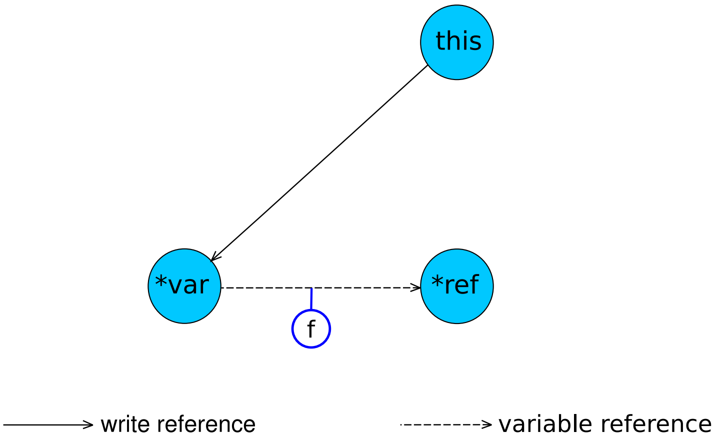

Figure 2.9: The Extended Object Graph after the field modification.

Obviously, the **this** object is performing a write operation on the object referenced by var (denoted $*$var). Therefore, a write reference is inserted between these two objects. It is not possible to identify the variable var as being used for this operation. Therefore, the write reference has no relation to any variable.

On the other hand the field being modified can be identified with a field modification event. So, it can be determined that the object referenced by var will store a reference to the object being referenced by ref in its field f. To capture this information, a variable reference is inserted in the Extended Object Graph which is related to the variable f.

Of course, the field ref has to have been written previously. Therefore, there is certainly also a variable reference between **this** and the referenced object. But this has nothing to do with the current operation, so it was omitted.

In this project the static fields where completely left out. It was considered coherent to leave the handling of static fields until the work on static methods is started.

Class preparation

The class preparation event is triggered when the Java Virtual Machine has finished loading a class but before it is initialised. The main usage for this project is that it allows to get a list of all fields and methods of only these classes, that are needed by the program under consideration.

The list of fields is needed to register interest in field modification events. The list of methods is used to parse the signature of the methods and determine which parameter or return value is of reference type and which of basic type.

Ignored events

There are a lot of other events a tracing/profiling program may register itself to. They are not all listed here as most are irrelevant to this project. There is only one that needs further explanation, why it is not used. It is the field access event.

The main reason why the field access event is ignored, is that it does not add new information. Every reference that will be stored in a given field will first have to be written to it. That is why every value of a reference type field will be known by tracking the field modification events.

A possibility would have been to try to relate write operations—like calls to non-pure methods—to the field that was used to refere to the target object. This could have been done by comparing the line numbers of a field access event and a subsequent method entry event. This was considered unreliable since method call statements can be spread over several lines. Furthermore, unrelated statements can occur on the same line of code. It would not be possible to determine if the method call was really related to the field access.

2.5.2 Build dominator tree upon Extended Object Graph

As described above, the first approximation for the owner of an object will be its direct dominator. Fortunately, this is exactly the output of the Lengauer/Tarjan [12] algorithm. Because, each object will store a reference to its owner, this will construct a tree structure upon the Extended Object Graph. This allows, not only to identify the direct dominator of an object but also to the next candidate, if the direct dominator cannot be the owner of an object for some reason.

Please note that the whole argument, why the direct dominator is a good approximation for the owner of an object, is based on the rule, that only the owner or a peer may perform a write operation on an object. For this reason, only the write references of the Extended Object Graph are used during the dominator algorithm. The variable references are not considered.

Dominator algorithm of Lengauer/Tarjan

Here is a short description of the Lengauer/Tarjan [12] algorithm[1]. Depending on which version is implemented it runs in $O(m \log n)$ or $O(m\alpha(m,n))$, where m is the number of edges and n the number of nodes in the flowgraph. The function $\alpha(m,n)$ is the inverted Ackermann function. Initially, a depth-first search DFS is performed, starting at the root. Each node is assigned a DFS number. This builds a DFS tree upon the flowgraph. For simplicity's sake, we do not distinguish the node from its DFS number.

The idea of the algorithm is to first compute an approximation by determining the *semidominator* for each node. A *semidominator* of a node x is an ancestor of x with the following properties:

$$\begin{aligned} sdom(w) &= \min\{v | there\ is\ a\ path\ v = v_0, v_1, ..., v_k = w\ such\ that \\ &\quad v_i > w\ for\ 1 \leq i \leq k-1\} \end{aligned}$$

To find the semidominators the graph is traversed in decreasing DFS number. The algorithm maintains a forrest by applying following operations:

- $LINK(v,w)$: v and w are root nodes of trees in the forrest. This operation adds the edge (v,w) of the DFS tree to the forrest.
- $EVAL(v)$: On the path from the root to v in the tree of the forrest to which v belongs, the node with the minimum key value is returned.
- $UPDATE(v,k)$: sets $key(v) = k$, where v must be a singleton tree.

First, for every node v the key is initialised with $key(v) = v$. Then the nodes are visited in decreasing DFS number. For every node v visited $UPDATE(v,k)$ is called where $k = \min\{EVAL(w)|(w,v) \in E\}$.

After the node v has been visited $LINK(v,w)$ is called for all children w of v.

At the end of this part of the algorithm $key(v) = sdom(v)$. The direct dominator can then be found in a further step within the same complexity. The algorithm iterates again over each vertex w. For each vertex v, where $sdom(v) = parent(w)$, the vertex u is found with $u = EVAL(v)$. The operation $parent(w)$ denotes the parent of vertex w in the DFS tree. The dominator of v is now either u, if $sdom(u) < sdom(v)$ or $parent(w)$ otherwise.

[1] This description is heavily based on the one given by Alstrup, Lauridsen and Thorup [14]

There may be some vertices, for which the direct dominator is still not yet computed. In the last step of the algorithm, for each vertex w where the dominator of w is not equal the semidominator of w, the dominator of w is set to the dominator of the dominator of w.

2.5.3 Store depth in dominator tree in each node

In the step after this one, a processing priority will be assigned to each write reference. The exact definition of this priority will be given there. However, in order to determine the priority, the depth in the dominator tree is necessary.

A naive approach to find this depth would be to perform a depth first traversal and to determine for each node encountered the objects that are owned by it. This would result in a $O(n^2)$ algorithm since for each objects, all other objects have to by inspected.

A better approach is given in pseudo-code:

Listing 2.3: Determining the dominator level of each node.

```
store dominator level 0 in root
put all objects in worklist
for each object in worklist {
        if (dominator level already stored) {
                remove object from worklist
        }else{
                follow owner reference until first object with stored dominator level
                assign dominator level to each object encountered
                remove object from worklist
        }
}
```

This yields a complexity of $O(n \log n)$. An alternative would be to directly remove an object as soon as the dominator level is assigned to it. But this would require to find it in the worklist, which usually takes $O(\log n)$ and would result in the same overall complexity.

2.5.4 Determine priority for each edge and build worklist

In this step the algorithm iterates over each edge and computes its priority, which is then stored in the edge. Here are again the two factors of the priority:

$$min\_dom\_lvl = min(A.dominatorLevel, B.dominatorLevel)$$

$$dom\_lvl\_diff = abs(A.dominatorLevel - B.dominatorLevel)$$

The *dom_lvl_diff* is first computed without applying the *abs*() function. If the sign of the result is greater than zero, it means that the edge connects a node from a larger depth in the dominator tree with a node from a smaller depth. That is the definition of a conflicting edge and it will be put on the worklist.

That way, in one step, the priority of each edge is determined and the worklist is built.

2.5.5 Resolve conflicts through processing of the worklist

The algorithm to process the worklist and resolve the conflicts has been described above. Here is the concise description in pseudo-code:

Listing 2.4: Conflict Resolution in pseudo-code

```
Pop edge with highest priority and mark as visited
Set owner of start to the owner of end
Store start as actual object
// entry point of recursion
Sort ingoing and outgoing edges of actual object
For each ingoing or outgoing edge of actual object that was not yet visited {
        mark edge as visited
        tmpobject := object at the other end of the edge
        if(actual object is neither peer nor owner of tmpobject){
                set owner of tmpobject to the owner of actual object
                if(worklist contains edge){ remove it }
                set tmpobject as actual object and recurse
        }
}
```

It is important to mark already visited edges. First, it is more efficient to check a boolean value on an edge than to compare the owners of two objects, like it is done in the if-condition inside the for-loop. Second, this ensures that every edge is really only processed at most once.

Not only the incoming edges are followed. In figure 2.10 when the large cycle A—B—C—D—E—A is resolved, all these nodes will become peers. From the dominator algorithm, Z is in the context owned by C and is therefore peer to D. But when the algorithm make D peer to C then the reference D—Z will become a conflict, since it will cross the boundaries of the context owned by C. This illustrates, why also outgoing references are followed.

2.5.6 Harmonise and annotate

This is the last step in which the annotation for each variable is inferred. This information can only be gained from the variable references. The write references are ignored in this

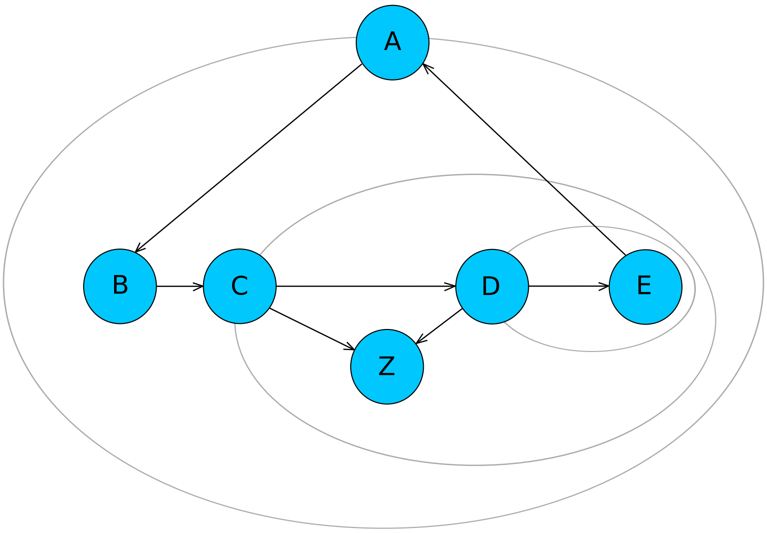

Figure 2.10: Outgoing edges have to be followed, too.

step, since they are not linked to any variable.

The Extended Object Graph of an already annotated program will have following property: If a variable is annotated as **rep** then all variable references, to which the variable is attached, will connect the owner to one of its owned objects. If the variable is annotated as **peer** then the corresponding variable references will always connect objects that are peer to each other. For the **readonly** annotation, nothing can be said about the relationship between the connected objects.

This may seem obvious, but it illustrates and motivates the goal, this algorithm must achieve. As stated above, the objects of the same class may be used differently in the same program and the variable references of the same variable may suggest a **peer** annotation in one point and a **rep** annotation in another. This has to be harmonised.

To begin the algorithm has to identify all variable references for each variable. It may happen, that all variable references of a variable connect an owner with one of its owned objects or that they connect only objects that are peer to each other. In such a case, no harmonisation has to be done and the variable can be annotated accordingly.

But for the case described above, we need a harmonisation. Either all objects of the references are made peer, or the start object is made owner of the end object for each reference. The latter is not possible, as the owner of an object has to be one of its dominators. Additionally, the end object was placed in its context because of incoming write references from other objects. All of these references would become conflicts if the context of the end object is changed.

The only possibility is by raising the end object to the context of the start object, thus making them peers. This is in principle the same approach as for the conflict resolution.

There is one important difference that makes the task simpler: There are no conflicting edges. That means that all incoming edges of an object start at a peer object or at the owner. Therefore, if an object is raised in the context of its owner, the reference from the same will not introduce new conflicts. All that needs to be done is to also raise the peers of the object, that are reachable from it. To determine, if an object is reachable or not, the direction of the edges is ignored. The idea is that if an object has no connections to one of the objects being raised, then no new conflicts will be introduced.

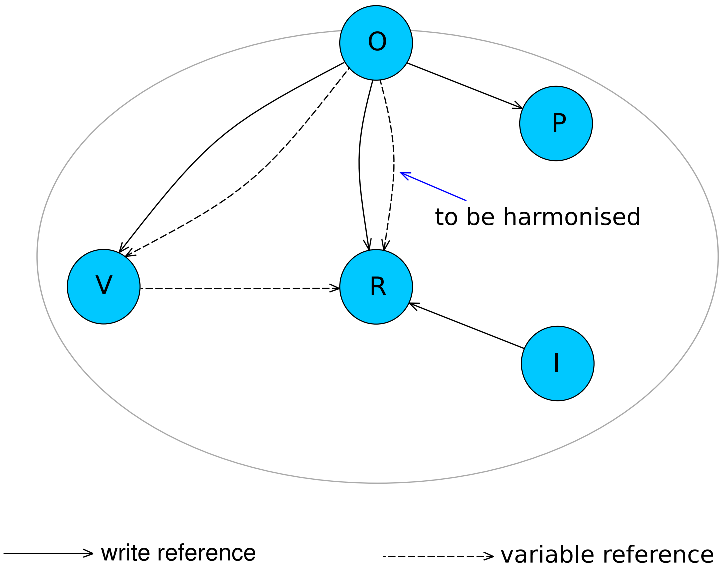

Figure 2.11: Interesting cases in the harmonisation phase.

In figure 2.11, object R is connected to its actual owner through a write reference that needs to be harmonised. Therefore, R has to be raised to the same context as O. The object I has only an outgoing reference to R that is going to be raised. It is still considered reachable, since the direction of the references is ignored. For this reason it will be raised, too. Object V is only connected to R through a variable reference. But if this reference is already annotated—peer in this instance—and R is raised then it will cross the boundary of the context owned by O and would be wrongly annotated. It is not desirable to develop an algorithm, that is correct only if by chance the variables are processed in the correct order. Therefore, peer objects that are linked with a variable reference, like V, have to be raised, too.

On the other hand, object P has no reference whatsoever to either R, I or V, that are being raised. So it is not necessary to raise it, too, since no conflicts can be introduced.

The previous owner (O in the example) may have variable references to objects that are also being raised, like the variable reference O—V. The corresponding variables may already have been annotated. The raising of these objects would invalidate these annotations. That is why all the variables of the class of O need to be processed again. The termination of the algorithm is still guaranteed, as all changes to the Extended Object Graph lead to a structure that is less deep. No changes are revoked.

Readonly Variable References

When a variable reference exists that connects two objects that are not in a peer or in a owner—owned relation, the corresponding variable can only be annotated with **readonly**. Regardless of what annotation the other variable references suggest. This is correct in the most cases, as methods with **readonly** parameters may also be called with objects that are peers of, or even owned by, the target object. A good example is the equals() method of the class Object in Java.

But problems may arise, in the following situation. A singly linked list is implemented in two classes: the List class which implements the list head and the ListItem which represent an element in the list, that will hold a reference to the stored object.

Listing 2.5: Singly linked list

```
class ListItem{
        ListItem next;
        Object stored;
        ListItem(Object toStore) {
                stored = toStore;
        }
        void insert(Object toStore) {
                if(next == null) {
                        next = new ListItem(toStore);
                }else{
                        next.insert(toStore);
                }
        }
        void remove(Object toRemove) {
                ...
        }
}
```

Not the list head inserts a new list item at the end of the list for a new element, but the element to store is handed down the list. The same is true for the removal of elements from the list, which is not shown in the code. If elements are only inserted in the list, the algorithm will find a **rep** annotation for the next field. But if an element is deleted from the middle of the list and no new element is inserted, then the predecessor of the deleted element will receive a variable reference to the successor in the Extended Object Graph. These two objects will not be connected by a write reference. The variable reference will therefore connect two objects that are neither peers, nor owner—owned related. The algorithm will have no other choice as to annotate the next field with **readonly**, which is clearly not precise enough and would require downcasts to be correct.

If on the other hand another object is inserted in the list after the object was deleted, then the predecessor would have a write reference to the successor. This would result in a false cycle in the Extended Object Graph, which would be resolved by making all list items peer to each other and would lead to a **peer** annotation for the next field, which is correct.

This problem can arise in all recursive structures. Clearly, the program execution determines if the corresponding variables will be annotated correctly or not. That shows that the solution to this problem lies in a good code coverage of the program under consideration, which is out of the scope of this project. The knowledge of this problem may help users to design good tests for their program to avoid this problem.

Variables, variable references and inheritance

The variables that are attached to variable references are always taken from the *declaring class*, not from the actual class. That way, when the algorithm has to identify all variable references of a given variable, all references are found. Including the ones, that start at instances of a subclass. An other advantage is that an additional harmonisation of annotation on different inheritance levels can be avoided. The found annotation is valid for all subclasses.

2.5.7 Generate output

At the end, the inferred annotations need to be written to a file. The format of this file is XML that conforms to the annotation.xsd XML-schema (see appendix A). It can be found together with the other project files.

Chapter 3

Implementation

3.1 Choice of Technology

The first problem concerning the implementation was the decision, which technology to use for the information gathering. We introduce first the different options and describe later, why which technology was chosen.

3.1.1 Java Platform Debugger Architecture

One of Sun's new JDK 5.0 "Tiger" highlights are its improved Java Platform Debugger Architecture (JPDA). It is composed of several different parts that provide different levels of abstraction and different functionalities.

Java Virtual Machine Tooling Interface (JVMTI) is at the lowest level. It is a C/C++ interface that defines a number of callback functions that are used by the Java Virtual Machine to pass information to an agent, and a set of interface functions that allow an agent to further inspect the Java Virtual Machine. Most callback functions have the purpose to inform an agent if a given event has occurred. Only the events are reported to which the agent has registered itself for. JVMTI agents are loaded by the Java Virtual Machine as shared libraries.

Java Virtual Machine Debug Interface (JVMDI) The JVMDI is an earlier version of the JVMTI with the sole purpose of providing an interface for debuggers. It works quite the same way, but is less powerfull. It is now deprecated and will not be supported in a future version of the Java Virtual Machine.

Java Virtual Machine Profiling Interface (JVMPI) The JVMPI, was an experimental feature of Java 2.0 SDK. As with the JVMDI, it provides only a subset of the JVMTI functions. It seems that the JVMDI and the JVMPI where merged together in the JVMTI.

Java Debug Wire Protocol (JDWP) Since the JVMTI allows only in-process tracing, a protocol is needed to allow out-of-process tracing programs to communicate with the Java Virtual Machine. The JDWP fills this gap. It defines packet formats for each event to be sent to the tracer and for each command a tracer can issue to the Java Virtual Machine.

On the side of the Java Virtual Machine an agent is needed that acts as communication partner for the JDWP. Such an agent is called *transfer* and is implemented as a JVMTI agent. A reference implementation is shipped with the JDK 5.0.

Java Debug Interface (JDI) is a Java library that uses the JDWP and the JVMTI to provide an easy to use tracing and profiling interface to be used from inside a Java program. It can be seen as a client side implementation of the JDWP.

3.1.2 Considered technologies

There are a lot of third party tracing and profiling tools but all of them are based on one or more of the technologies of the JPDA. Following is a list of technologies that where considered, with an assessment of the pros and contras for this project:

JVMTI Since the JVMTI is the lowest level of the JPDA it is the most powerful. If a functionality is not present, it can be inserted by the means of bytecode instrumentation. This interface provides functions to implement this task. A disadvantage is that it only allows in-process tracing which makes it difficult to interact with a profiling program outside the Java Virtual Machine that is running the program to trace. The biggest disadvantage is, that it is very inconvenient to program with. Even the simplest tasks need several lines of code to be implemented.

JVMDI/JVMPI Both where not considered suitable as they are deprecated. Furthermore, they do not provide enough information. For instance, it is not possible to get information about method return values.

JDWP To program a client side implementation would have been a good possibility to overcome the shortcomings of the JVMTI as it allows out-of-process tracing and profiling. Unfortunately, it is not yet fully adapted to the new JVMTI interface. The method return values, that are provided by the JVMTI in method exit events, are not defined in the corresponding JDWP packet format.

JDI The limitations of the JDWP are propagated to the JDI, which is based on the JDWP. Here too, it is not possible to find out the method return value.

Eclipse Test and Performance Tools Platform Formerly known as Hyades. It provides a fully in Eclipse integrated tracing application. Dave Smith of the IBM Toronto Laboratory confirmed, that it does not provide information about field modification, method parameter values and method return values.

Eclipse JDT Debug We also looked at the Java debugger of Eclipse. It is based on JDI and can therefore not be more powerful.

JSwat JSwat is a free Java debugger. Unfortunately, it is also based on JDI.

3.1.3 Decision

Obviously, the preferred technology would have been JDI. It provides the simplest interface and allows to start a second Java Virtual Machine to trace a program. Because of the limitations mentioned above it was considered not suitable for this project. Maybe in a future version, when JDI is adapted to JVMTI, it would be worthwhile to rewrite the tool created in this project in JDI.

It turned out, that only the JVMTI provides enough information for this project. Specially, the method return value cannot be determined with other technologies.

3.2 Architecture

Because it is very cumbersome to program with JVMTI, we wanted to reduce this part to a minimum. The main part of the algorithm should be implemented as a Java program. In order not to interfer with the program that needs to be traced, the Java program for the type inference has to be run in another Java Virtual Machine. As stated above, a JVMTI agent runs from within a Java Virtual Machine. That means that it can not use JNI's invocation API as only one Java Virtual Machine can run inside one process.

The simplest solution to this problem is to separate the task in two parts. The first part consists of a JVMTI agent that is loaded from the Java Virtual Machine that runs the program that has to be annotated. The agent does nothing else, than collecting the necessary information and writing it to a file. This reduces the part that has to be written in C++ with the JVMTI to a minimum.

The second part consists of a Java program that runs *after* the first part. It reads the file with all the information, runs the type inferring algorithm and outputs the annotations found.

The separation of the information gathering and the algorithm has another advantage. In his semester project [15] Marco Meyer writes a visualisation of this algorithm. This visualisation should be of the form of an editor, that allows to go through the algorithm step by step and to modify the Extended Object Graph or the algorithm on the fly. This means that a user may want to run the algorithm on the same information several times and to alter her input to the editor. In such a case it is not necessary to retrace the same program with the same input over and over. It is much more convenient to trace the program once and to reuse the gathered information.

This leads to the architecture that is represented by figure 3.1.

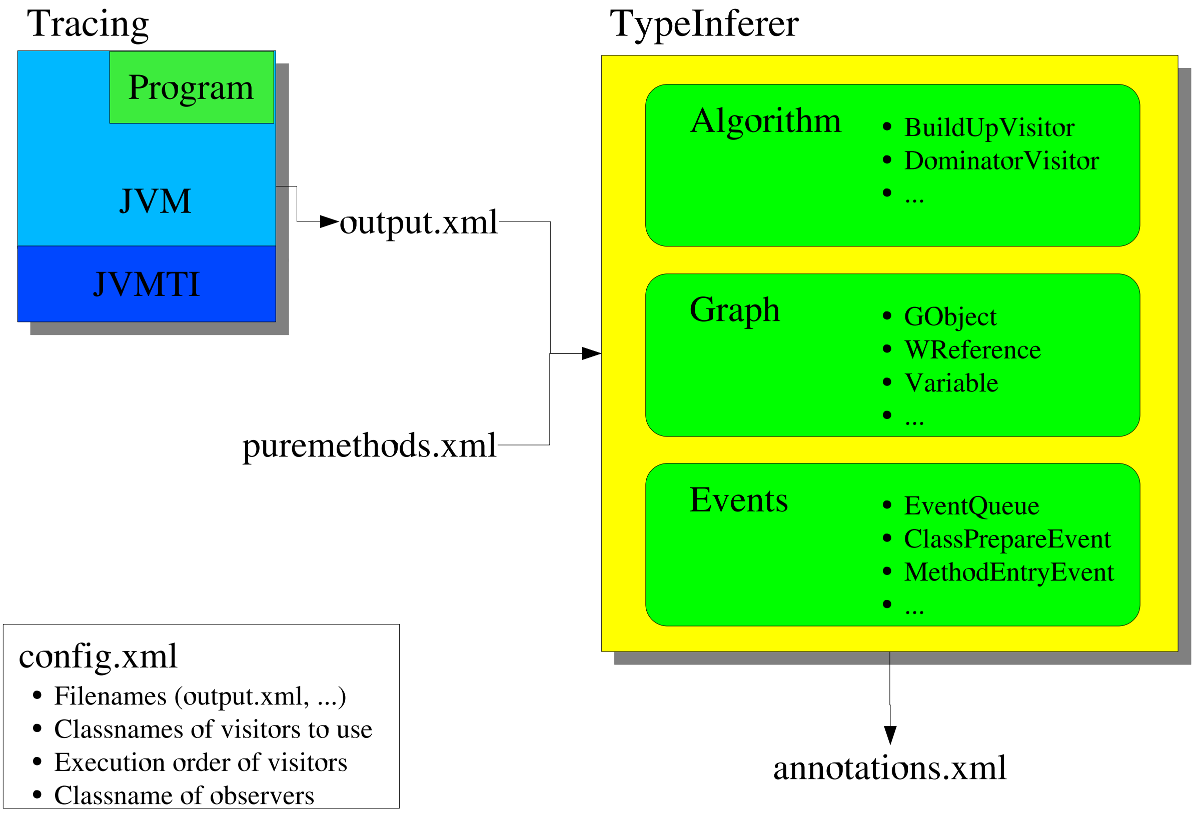

Figure 3.1: The overall architecture.

3.2.1 JVMTI agent

As described above, the JVMTI agents sole purpose is to gather the required information. For each event, the agent compiles the information and prints it to the output file in a XML tag structure. In that way, the type inferring program will be able to reconstruct each event in the order in which they occurred and process them accordingly. The output XML file conforms to the agentoutput.xsd XML schema which can be found in appendix B.

The objects are identified by a **long** value. The agent has a counter which is incremented for each <init> method that is called on a new object. The JVMTI allows to tag objects with a **long** value. This feature is used to store the identifier of a given object.

3.2.2 Type inferring program

The type inferring program is written in Java. We used the SDK 5.0 to be able to use the new language features. The type inferring program has no dependencies on Multi Java or JML, the two platforms on which the Universe type system was implemented. It consists of four packages:

graph package

The graph package contains all classes needed to implement the Extended Object Graph. Figure 3.2 is the UML representation of this package.

Representation choice

Since the conflict resolution involves the search for cycles, we considered storing the graph in an adjacency matrix. The corresponding algorithm to find all possible cycles has a complexity of $O(n^4)$, which is quite inefficient. However, Potanin and Noble [16] in a large study, showed that on average only 12.86% of the objects in a Java program have more than one incoming reference. That means that storing object graphs in an adjacency matrix leads to an extremely sparsely populated matrix.

The alternative we chose, was to use separate objects to represent the references. The start and the end object then have a reference to such a reference object. Additionally, the reference object has itself references to the start and the end object. Please, see figure 3.3 for further illustration. This solution has the advantage, that only memory is allocated, that is really needed. The disadvantage is that for every reference a new object is allocated and four references are stored: one to each associated object and one from each associated object. That means that to store one reference 4 times 4 = 16 bytes are allocated beside a new object. The total memory consumption is still a lot smaller than the one of the first alternative.

GObject class

This class corresponds to an object of the Extended Object Graph. It stores an

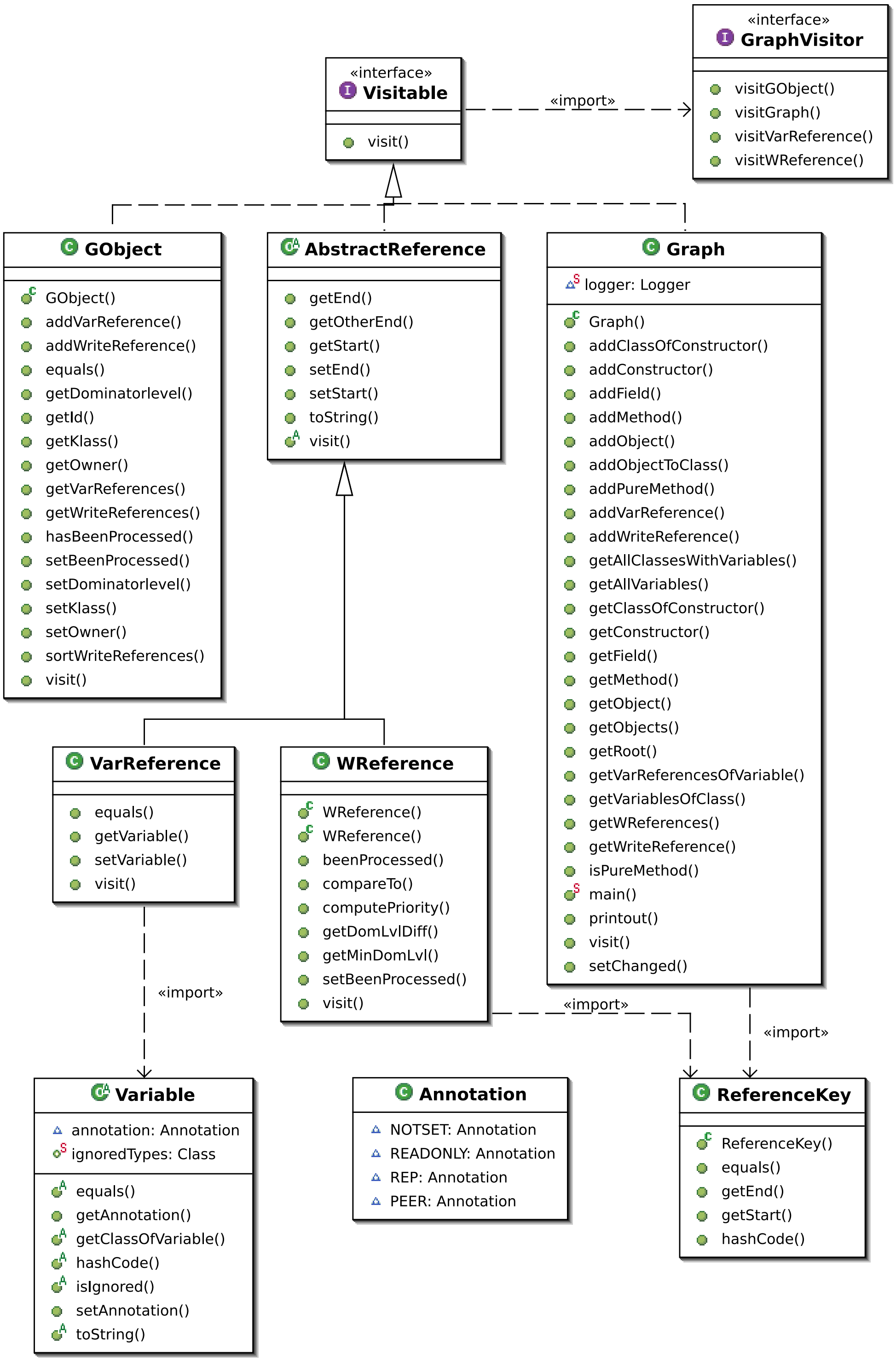

Figure 3.2: UML diagram of the graph package.

identifier, a reference to its owner, its dominator level and the class of the object. Additionally, there is a collection of all WReferences and a collection of all VarReferences.

WReference class

Objects of this class correspond to write references. As can be seen in figure 3.3 a WReference has a reference to the start object and a reference to the end object. Additionally, both object have a reference to the WReference. This allows to freely navigate in all necessary directions.

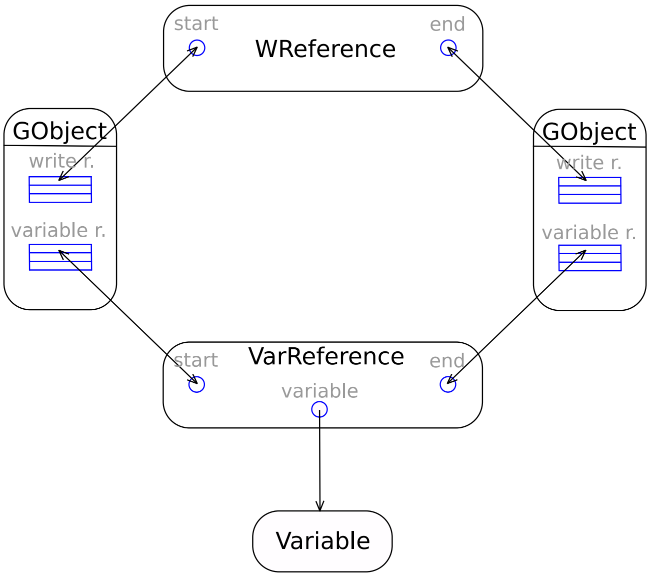

Figure 3.3: Write and variable references.

VarReference class

Not surprisingly, objects of this class correspond to variable references. They have nearly the same features as the WReferences. Additionally, they store a reference to the variable in which the reference was stored. This is shown in the lower part of figure 3.3.

Variables

Three kinds of variables exist: fields, method return values and method parameters. As can be seen in figure 3.4, they are represented by the classes FieldVariable, MethodVariable and ParamVariable, respectively. Each of these can store the inferred annotation and a reference to the corresponding reflection object. That would be a `java.lang.reflect.Field` in case of a FieldVariable, a `java.lang.reflect.Method` in case of a MethodVariable. In case of a ParamVariable, the method and the parameter number are stored. The number of the parameter has to be taken as identifier, since a subclass can overwrite the method and change the names of the parameters.

Generally speaking, all type information that is used in this project is based on the reflection API. This is simpler to handle in the program, but the user should be aware that the program to be annotated needs to be in the classpath of the type inferring program.

Graph class

This is the main class of this package. It consists of a collection of objects, a collection of write references and a collection of variable references.

References of any kind are added by providing the start and the end object. This way, it is ensured, that the newly created reference is also correctly stored in the start and/or end object.

events package

The classes in the events package are responsible to read in the file that was written by the JVMTI agent. For each event type there is a class that defines the fields and the appropriate getter and setter methods to store all information gathered for the given event.

The prepared event objects are placed in an event queue. This preserves the order of the events and allows to run the reading/parsing task and the Extended Object Graph build up task in two different threads.

To read-in the file and parse the XML, Jakarta's Commons Digester [17] is used. It is simple to use and highly customisable. The event queue is built from a BlockingQueue and the digester runs in a different thread. This allows to control the amount of transient event objects that are kept in memory. If the event queue is full, the digester simply stops to parse the file until there is room in the event queue.

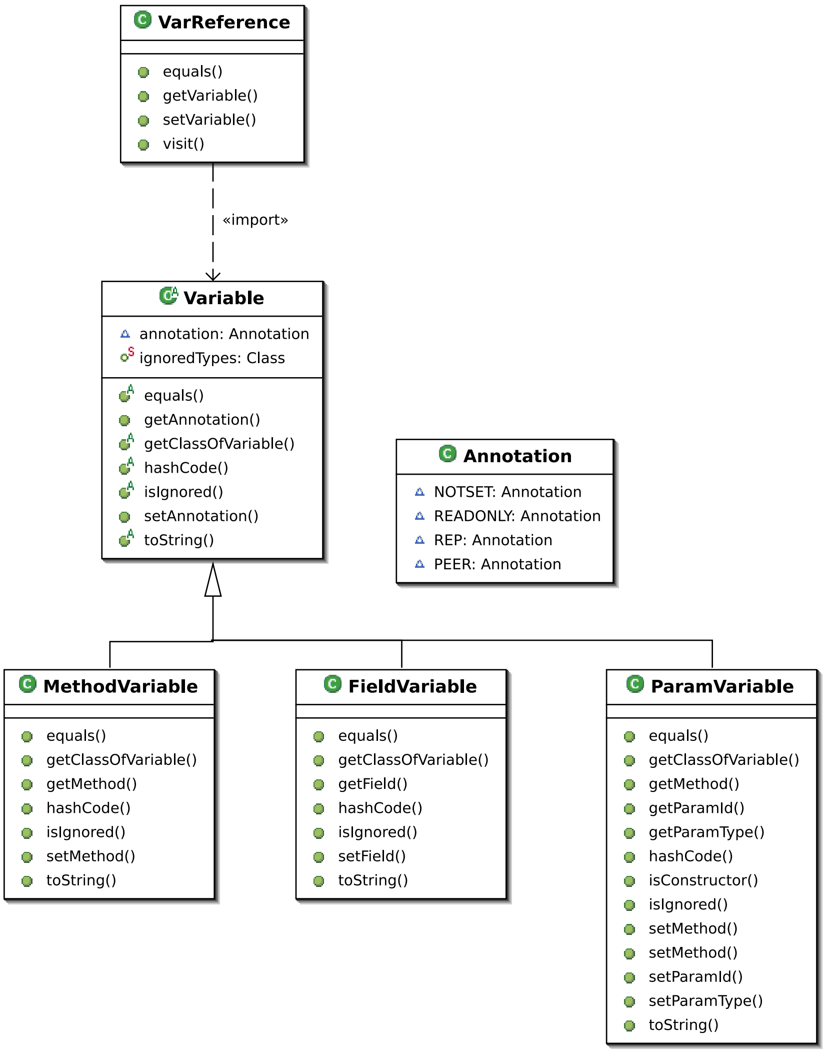

Figure 3.4: The three different kinds of variables are implemented as subclass of the generic Variable class. The class Annotation is an enumeration that defines the possible values of a variable annotation.

algorithm package

The algorithm is implemented in a class for each step. Each of these classes implement the GraphVisitor interface. They operate directly on the graph using the Visitable Interface that is implement by each graph entity. Here is the description of each of the implemented classes:

BuildUpVisitor Surprisingly, the graph is built up by a visitor. The BuildUpVisitor operates on an object of the type Graph that as no GObjects or references stored.

It initialises the event queue, which reads in the information from the JVMTI agent.

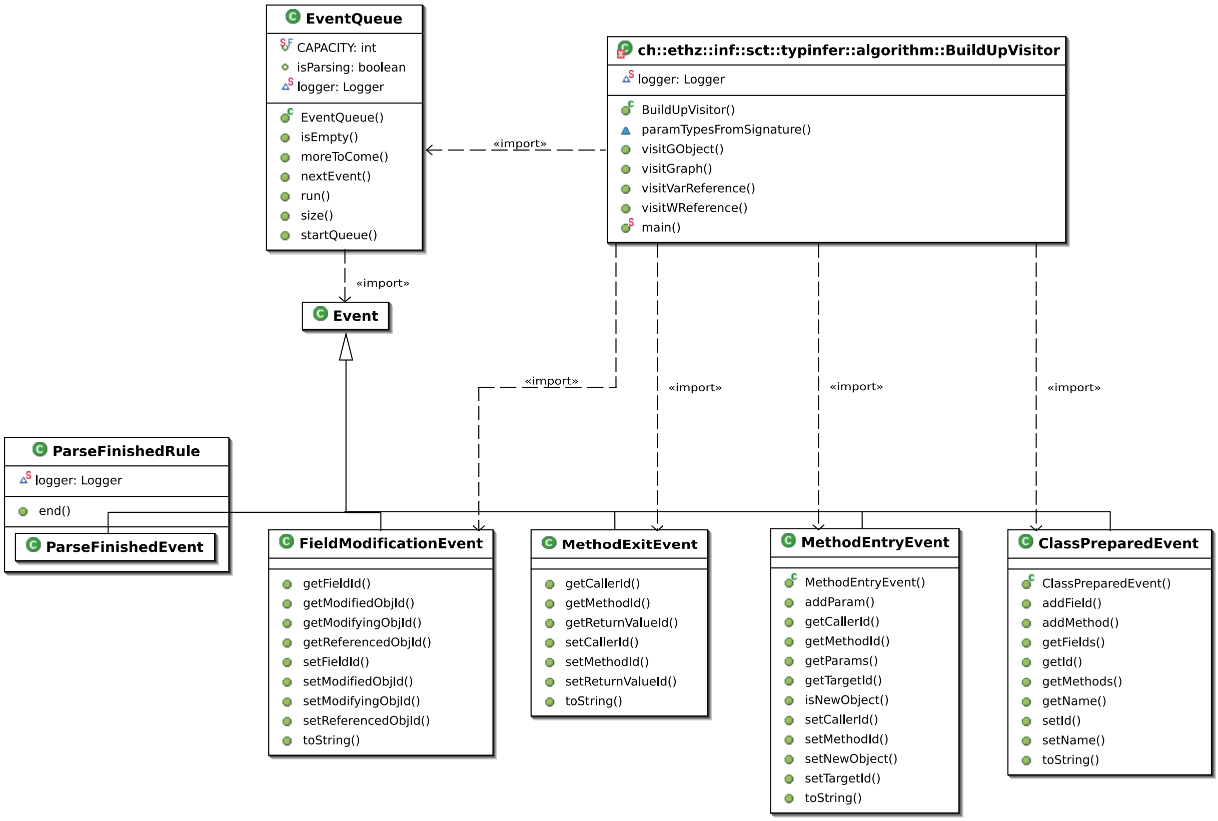

Figure 3.5: UML diagram of the events package.

Each event is processed as described in section 2.5.1.

DominatorVisitor After the graph has been built up, the dominator algorithm is performed upon it. This computes the direct dominator for each object and the **owner** field of every GObject is set.

StoreDominatorLevelVisitor This visitor computes the depth of each object in the dominator tree and stores this information in each object.

ResolveConflictsVisitor The conflicts introduced by the approximation are then resolved by this visitor. This visitor uses the Worklist class which is in fact an adapted PriorityQueue.

HarmonizationVisitor Then the dynamic structure of the Extended Object Graph is mapped to the static structure of the program classes. Inconsistent references are harmonised and the annotations are found for each variable.

OutputVisitor Finally, the found annotation is written to a XML file that conforms to

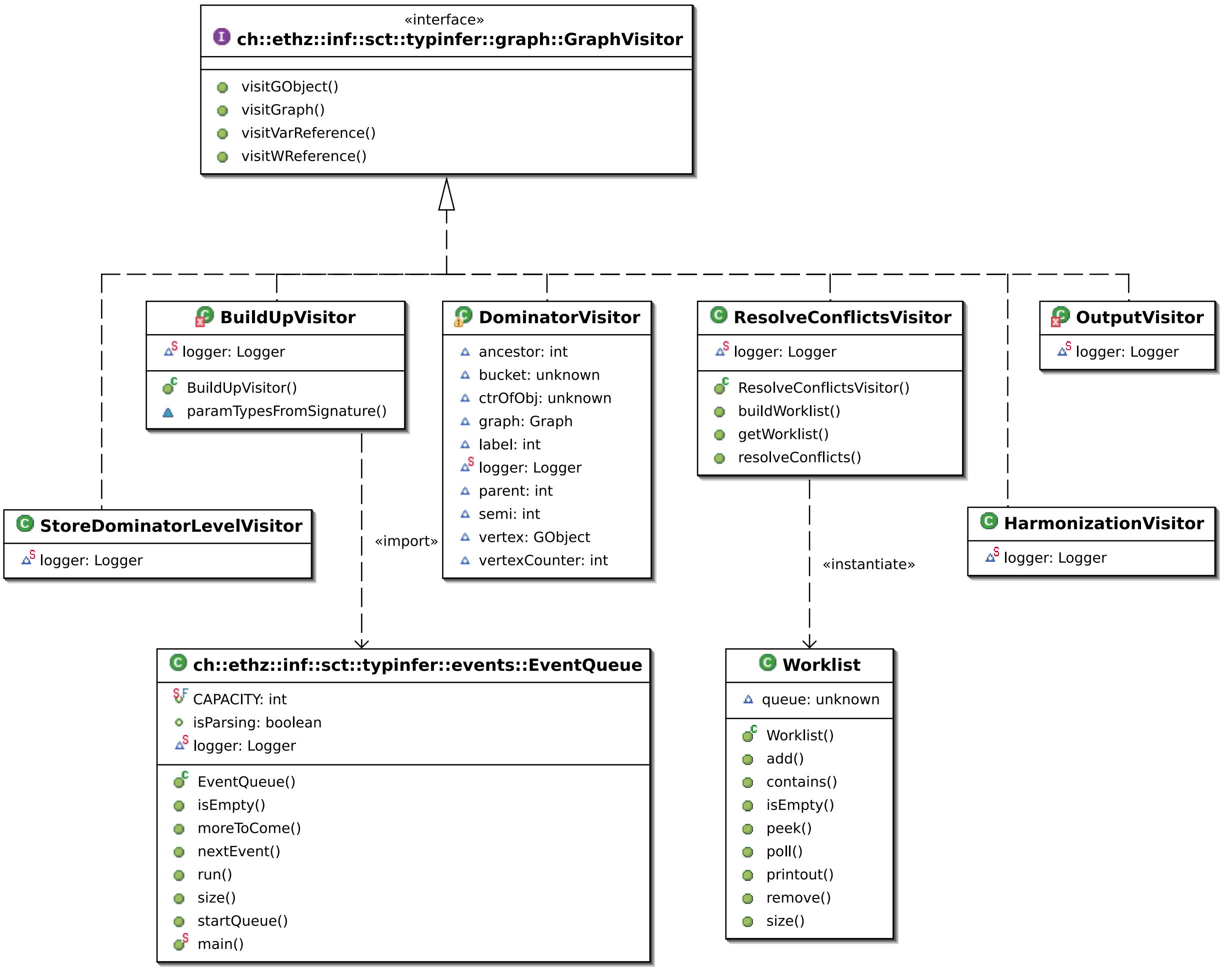

Figure 3.6: UML diagram of the algorithm package.

the annotations.xsd schema[1].

Root package

As shown in picture 3.7 the root Package contains only two classes. The Configuration class implements the Singleton pattern [18]. The static method getConfiguration() parses the config.xml file if necessary and returns the only possible object of this class. This allows any interested part of the program to get access to the configuration without the need to store the reference to it at all places.

The TypInferer class contains the main method of the type inferring program. The main method calls the method runAlogrithmFromConfiguration(). This method gets the con-

[1] See appendix A

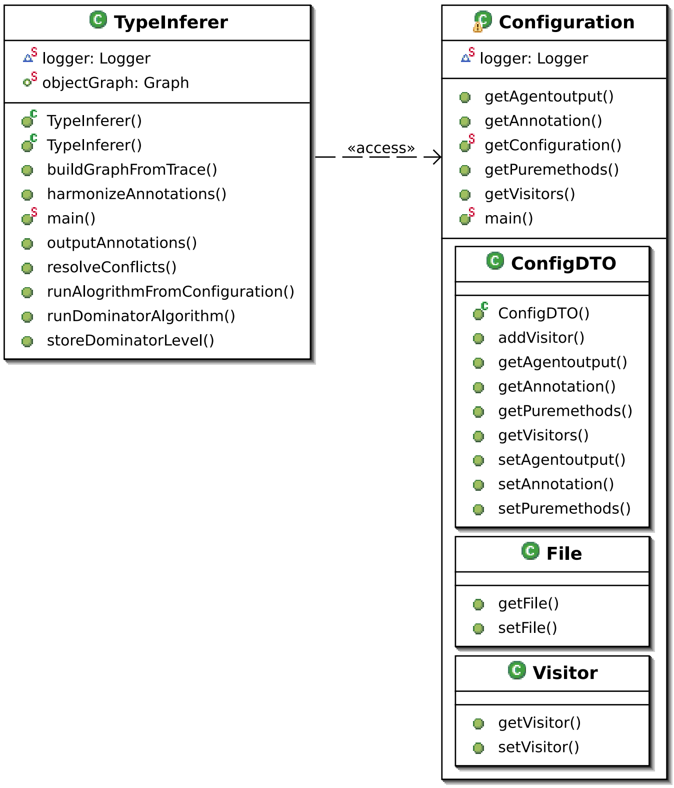

Figure 3.7: UML diagram of the root package.

figuration object, instantiates all observers and visitors and runs them.

Alternatively, a client program could instantiate the TypInferer class and some custom visitors and run them using the runVisitor () method.

3.2.3 Pure methods

As stated before, it is not part of this project to determine which methods are pure and which are not. The user has to provide the list of methods that are pure. This list has to be written in a XML file that conforms to the XML schema annotation.xsd. In figure 3.1 this is represented by the filename puremethods.xml.

3.3 Observer Interface

The algorithm visualisation described on page 47 was an important motivation to provide an interface that allows to observe each state change of the graph. It was decided to implement this interface using the Observer pattern [18].

The pattern usually uses a generic notification method to inform the registered observers that a state change has occurred in the observed system. This is not precise enough for our application, since changes can happen at different levels: at the object/reference level and at the graph level. An instance of a object/reference level change is when an object is marked as *been processed*. Changes on the graph level are when objects or references are added to the graph.

For this reason a set of notification methods have been defined. Here is the complete GraphObserver interface with a description for each method:

newObjectAdded(GObject, GraphVisitor): This method is used to notify the observers that a new object was added to the graph.

newVarReferenceAdded(VarReference, GraphVisitor): Notification that a new VarReference was added to the graph.

newWReferenceAdded(WReference, GraphVisitor): Notification that a new WReference was added to the graph.

objectBeenProcessedChanged(GObject, GraphVisitor): Notification that the beenProcessed field of the given object has changed.

dominatorLevelChanged(GObject, GraphVisitor): Notification that the field dominatorLevel of the given object has changed.

ownerChanged(GObject, GraphVisitor): Notification that the **owner** field of the given object points to a new object.

wRefBeenProcessedChanged(WReference, GraphVisitor): Observers are notified that an edge was processed.

priorityChanged(WReference, GraphVisitor): Notification that the priority of a WReference has changed.

annotationChanged(Variable, GraphVisitor): Observers are notified when the annotation of a Variable has changed.

Every method in this interface takes a GraphVisitor as the second parameter. The reason is that it may be important for a given observer to know which GraphVisitor produced the changes.

The Graph class extends the GraphObservable class. This class provides a method addObserver() that allows to register a new observer to the graph. Additionally, it provides for each method xxx() described above a corresponding notifyXxx() method. For instance the annotationChanged() method corresponds to notifyAnnotationChanged(). It is the responsibility of the GraphVisitors to call the appropriate notification method on the graph when they change its state.

All observers are notified with synchronous calls in the main thread. This automatically stops the algorithm until the observers have finished the processing of the actual notification. This is done to allow the implementation of observers that allow the user to step through each action of the algorithm. Additionally, this does not introduce any restrictions as every observer is free to start its own thread and process the notification asynchronously.

3.4 Configuration

As shown in figure 3.1 there are three files involved in the process. The one that is called output.xml in the picture is the file used to stored the gathered information from the JVMTI agent. This file is then read by the type inferring program. The inferring program also reads in the list of pure methods. And finally, after the algorithm is terminated the annotations are printed out in a file that is here called annotations.xml.

For a more convenient use, these filenames can be specified in a file called config.xml. Additionally, the algorithm itself can be configured. The config.xml file provides a list of classnames of the GraphVisitors that make up the algorithm. The type inferring program instantiates and executes these GraphVisitors in the order in which they are listed.

In order to use the Observer Interface described in section 3.3, the observers to be instantiated can be specified in the config.xml like the GraphVisitors. For both, the GraphObservers and the GraphVisitors, it is checked if they implement the corresponding interface. If not, an error is printed and the program is terminated.

There are only two files that need to be modified by the user. One is the config.xml file that is used for the overall configuration and the XML file that lists the pure methods. The XML schema of the config.xml[2] is straight forward. The annotations XML schema[3] is much more complicated. For the list of pure methods only a small subset of the schema elements are required.

The <head> tag should be present for documentation reasons. Each class that contains a pure method has to be listed and has to contain the corresponding method tag. Please note that the parameters have to be listed. Otherwise, the program is not able to find the

[2] see appendix C

[3] see appendix A

corresponding method object and will crash. The tag for the return value is not required to be present.

If the pure methods file could not be loaded, please make sure, that the file is at the location configured and has the correct name. If that is already the case, the most likely reason for this error is that the file does not conform to the annotations XML schema or is not a valid XML file because of syntax errors.

3.5 Usage

3.5.1 JVMTI agent/information gathering

The usage of the JVMTI tracing agent is as follows:

```
java -cp <classpath> -agentpath:<ap>=<mc>,<of> <mc>
```

where:

classpath The classpath of the program to annotate.

ap The path to the JVMTI agent is represented as a shared library. This must contain the absolute path to the agent plus the agent name. On the linux computer on which the tool was written, the following was used: `/home/frank/eclipse/uts_type_inferer/jvmti/uts_type_inferer.so`

mc The fully qualified name of the class containing the main method to execute. This includes the exact package name of the class. Of course, this class has to be in the classpath specified above.

This information is needed twice: first, the Java Virtual Machine has to know which program to start and second, the agent has to be informed about the main class to be able to start the tracing correctly.

of The path to the file into which the gathered information will be written. It is necessary that this path matches exactly the one given in the agentoutput-tag in the configuration file (see appendix C). In the architecture overview on picture 3.1 this file is called `output.xml`.

3.5.2 The type inferring program

The usage of the type inferring program is as follows:

```
java -cp <classpath> ch.ethz.inf.sct.typinfer.TypeInferer
```

The classpath *must* contain all classes of the program under consideration. Here is the list of libraries that where used for this project that also need to be referenced in the classpath:

- commons-beanutils-1.6.1.jar
- commons-collections-2.1.1.jar
- commons-digester-1.5.jar
- commons-logging-1.0.4.jar
- commons-logging-api-1.0.4.jar
- jaxp-1.2.jar
- jsr173_api.jar
- log4j-1.2.8.jar
- xercesImpl.jar
- xmltypes.jar
- xbean.jar

All of them are in the `lib` directory of the project files. Please note that the xmltypes.jar file was created from the annotations.xsd XML schema using the `scomp` tool shipped together with the XMLBeans library.

The configuration file described in section 3.4 has to be named config.xml. It has to be placed in the current working directory in which the program is started.

Chapter 4

Results and future work

4.1 Discussion of some examples

In this section we show some examples. First the program code to be annotated is given. Then the Extended Object Graph is shown, that is built by the algorithm. We then present the annotations found by the program and discuss them if necessary. For each example the memory and time consumptions is shown for the tracing and for the type inferring part.

4.1.1 Linked List

Listing 4.1 shows the code for a singly linked list. It consists of two classes: one for the list with the interface methods and one for the list items, that form the representation of the list. Please note, that not the list creates new list items to store new objects, but the last item in the list does.

Code

Listing 4.1: Code for singly linked list.

```
public class LinkedList {
        ListItem head;
        int size;

        public LinkedList() {
                head = null;
                size = 0;
        }
        public int size() {
                return size;
        }

        public void insert(Object o) {
                if(head == null){
                        head = new ListItem(o);
                }else{
                        head.insert(o);
                }
                size++;
        }

        public boolean contains(Object o) {
                if(head != null){
                        return head.contains(o);
                }else{
                        return false;
                }
        }

        public Object remove(Object o) {
                if(head == null){
                        return null;
                }else if(o.equals(head.stored)){
                        Object ret = head.stored;
                        head = head.next;
                        size--;
                        return ret;
                }else{
                        Object ret = head.remove(o);
                        if(ret != null){
                                size--;
                        }
                        return ret;
                }

        }
}

public class ListItem {
        Object stored;
        ListItem next;
        public String name;
```

```
    ListItem(Object toStore) {
        stored = toStore;
        next = null;
    }

    public void insert(Object toStore) {
        if(next == null){
            next = new ListItem(toStore);
            next.name = ”item”;
        }else{
            next.insert(toStore);
        }
    }

    public ListItem getNextItem(){
        return next;
    }

    public Object remove(Object o) {
        if(next == null){
            return null;
        }else if(o.equals(next.stored)){
            Object ret = next.stored;
            next = next.getNextItem();
            return ret;
        }else{
            return next.remove(o);
        }
    }

    public boolean contains(Object o) {
        if(o.equals(stored)) {
            return true;
        }else{
            if(next == null){
                return false;
            }else{
                return next.contains(o);
            }
        }
    }
}
```

Listing 4.2 shows the first usage of the LinkedList.

Listing 4.2: First usage of the linked list.

```
public static void main(String[] args) {
        LinkedList list = new LinkedList();
        A a1 = new A();
        A a2 = new A();
        B b1 = new B();
        B b2 = new B();
        list . insert (a1);
        list . insert (b1);
        list . insert (a2);
        list . insert (b2);
        list .contains(a1);
        list .contains(b1);
        list . size ();
        list .remove(a2);
        list .remove(b1);
        list . insert (a2);
        list . size ();
}
```

Discussion

This program is first traced. Then the Java program is started that infers the Universe types. The methods Object.equals() and ListItem . contains() are declared as **pure**.

The type inferring program builds the Extended Object Graph shown in figure 4.1. Only the variable references are shown that connect ListItem objects with the objects they stored. The number near each node is the id that the tracing program associated to each object. It corresponds to the order in which the objects where created.

The program performs the dominator algorithm on this graph. The root object 0 is found to be the direct dominator of the objects 1, 2, 3, 4 and 5. These objects are created in the main method and belong therefore to the root context. The list object 1 is the direct dominator of the item object 6 that stores a1. 6 is the direct dominator of 7, 7 of 8 and 9 of 10. Because of the edge 6—9, 6 is found to be the direct dominator of 9. The edge 6–9 is introduced because the objects 7 and 8 are removed and object 10 is inserted. This means that 6 is calling the insert () method on 9. This is a non **pure** method and therefore a write reference is inserted between 6 and 9.

The algorithm step that computes the dominator level for each node assigns level 2 for node 6, level 3 for node 7, level 4 for node 8 and level 3 for node 9. The next algorithm step, that assigns the priority to each edge and builds the worklist, finds that the edge 8—9 is a conflicting edge. Therefore, this edge is put on the worklist.

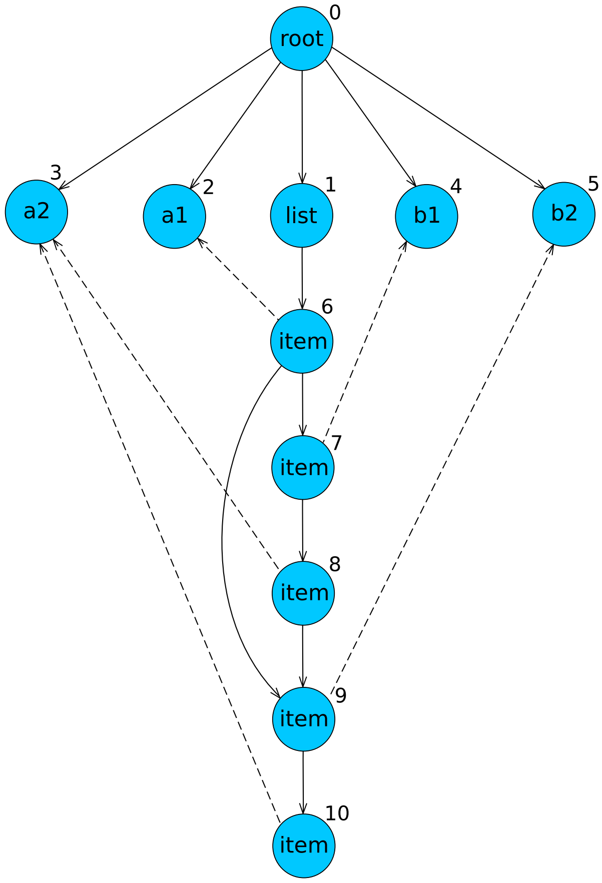

Figure 4.1: The Extended Object Graph built by the program from the tracing information.

The conflict resolution phase then follows the incoming edges 7—8 and 6—7 and makes the objects 7 and 8 peer to object 9.

All the ListItem objects are connected to each other through references stored in the next field. The harmonisation step of the algorithm compares the start and the end object of each reference stored in a next field. It finds that the objects of the references 6—7, 6—9, 7—8 and 8—9 are all peer to each other and that 9 is the owner of 10. This has to be harmonised by making 10 peer to the other ListItem objects. Now the next field can safely be annotated with **peer**.

All references ever stored in the stored field of a ListItem object connect two objects that are not at all in the same context. This is also true for the parameters of the methods insert (), remove(), contains() and the constructor of ListItem. All these variables are therefore annotated with **readonly**.

For the class LinkedList the methods insert (), remove() and contains() are only invoked with parameters that are peer to the list object 1 in this example. Therefore, the program annotates all these parameters with **peer**. This is obviously not the case in general. This demonstrates, that the input values that are used for the program execution can have a great impact on the quality of the annotations found.

Another example for this sensitiveness can be shown by altering the code in listing 4.2 just a bit. If the last insert statement `list . insert (a2);` is omitted, then the write reference 6—9 does not exist. That means that the dominator algorithm will assign 8 as the direct dominator of node 9 and no conflicting edge occurs. But the variable reference 6—9 is still existent, since the reference to 9 is assigned to the `next` field of 6 after 7 and 8 are deleted. The two objects of the variable reference are not owner one of the other nor are they peer. This variable will therefore have to be annotated with **readonly** which is clearly not desirable.

Conclusion

The example showed that the program infers the Universe types as expected. But it also showed that the quality of the inferred types is strongly dependent on the input values and usage of the program under consideration. A good rule of thumb is that—especially for recursive structures—it is important that all possible operations are performed in different order to get the best results.

Memory and time consumption

Running this example and measuring using the `time` command under linux produced the results shown in table 4.1. The left column shows the time measured from the program run without tracing and the right column with tracing with the JVMTI agent developed in this project.

	Normal	*JVMTI*
real	0.656s	1.416s
user	0.075s	0.834s
sys	0.027s	0.138s

Table 4.1: Time consumption with and without JVMTI agent.

Considering only the classes of the program, 10 objects where allocated that used 168 bytes of memory. The shared library that represent the JVMTI agent and has to be loaded into memory has a size of 143K. We do not expect that it introduces significant additional overhead, since it produces only one temporary object at a time and deallocates it right away.

The JVMTI agent traced 137 events. The method of the type inferring program that is directly started form the main method and which executed the whole algorithm takes

Step of Algorithm	*Time needed*
Build EOG	580ms
Dominator	4ms
Build Worklist	7ms
Resolve Conflicts	2ms
Harmonisation	23ms
Output	157ms

Table 4.2: Time consumption of each step of the algorithm.

2.021s. Table 4.2 shows how long each step of the algorithm takes. Please note that the build-up phase includes reading the agent output file which takes 0.481s. Not included are the reading and parsing of the configuration file which takes 0.001s and the reading of the pure methods file which takes 0.714s. The output phase includes the writing of the XML file that takes most of the time.

Listing 4.3 shows a larger usage of the linked list. This example is used to explore the memory and time consumption of a larger program.

A method was added to the list implementation that allows to remove objects at a given index in the list. By removing a series of subsequent objects from the list and inserting new ones after that, conflicting edges are introduced in the list.

Listing 4.3: The code of the larger example

```
public static void main(String[] args) {
        LinkedList list = new LinkedList();
        for(int i=0; i<1000; i++){
                if(i%2 == 0){
                        list.insert(new A());
                }else{
                        list.insert(new B());
                }
        }
        for(int i=0; i<400; i++){
                if(i%10 != 0){
                        if(i%21 == 0){
                                for(int j=0; j<(i%10); j++){
                                        list.remove(i);
                                }
                        }
                }else{ list.insert(new A()); }
        }
}
```

	Normal	*JVMTI*
real	115ms	6m43.257s
user	90ms	3m17.747s
sys	16ms	3m10.119s

Table 4.3: Time consumption of the tracing with and without JVMTI agent.

Step of Algorithm	*Time needed*
Build EOG	899.399s
Dominator	1.830s
Build Worklist	0.912s
Resolve Conf.	0.068s
Harmonisation	11.724s
Output	0.421s

Table 4.4: Time consumption of each step of the algorithm.

The type inferring program took overall 20min 16.306s to run. 593'558 events where processed. The agent output file which stored the events had a size of 79 megabytes. Table 4.4 shows the time consumption of each step.

Table 4.4 shows that by far the most time is needed to read in the generated events from the agent output file. This could be nearly eliminated by reimplementing the information gathering component in JDI. This component could then directly be used by the BuildUpVisitor to build up the Extended Object Graph.

For measuring the memory usage, the JMP [19] tool was used. A particularity of the algorithm is that it builds up a big datastructure in the beginning and does only add very few new objects after that. Several snapshots after the buildup phase have shown that on average 618'779 objects where allocated and used 13.58 Mb of memory. The greatest part of this was used by the 531'960 instances of the VarReference class that used 12.18 Mb of memory.

4.1.2 Lottery

Listing 4.4 shows the code for a lottery game. A person plays by obtaining a LotteryTicket and choosing a number. After the draw of the lottery, the person can get the winning number and check if the chosen number wins.

Listing 4.4: Code for the lottery game

```
public class LotteryGame {
        public static void main(String[] args) {
```

```
        Lottery lottery = new Lottery();
        Person person = new Person();
        person.play( lottery );
        lottery.draw();
        if(person.check()){
            System.out.println (”I’ve␣won!”);
        }else{
            System.out.println (”I’ve␣lost!”);
        }
    }
}

public class Lottery {
    WinningNumber winner;
    public LotteryTicket getNewTicket() {
        return new LotteryTicket();
    }
    public void draw(){
        winner = new WinningNumber();
        winner.number = 123456;
    }
    public WinningNumber getWinner() {
        return winner;
    }
}

public class WinningNumber {
    int number;
    public boolean isWinner(LotteryTicket ticket) {
        if( ticket.chosenNumber == number) {
            return true;
        }else{
            return false;
        }
    }
}

public class Person {
    LotteryTicket ticket;
    Lottery lottery;
    public void play(Lottery lottery) {
        this.lottery = lottery;
        ticket = lottery.getNewTicket();
        ticket.chosenNumber = 123456;
    }
    public boolean check() {
        return lottery .getWinner().isWinner( ticket );
    }
}

public class LotteryTicket {
    int chosenNumber;
}
```

Listing 4.5 shows the method and field signatures of the program together with the found annotations.

Listing 4.5: Code for the lottery game with annotations

```
public class Lottery {
    /*@ rep @*/ WinningNumber winner;
    public /*@ peer @*/ LotteryTicket getNewTicket();
    public /*@ rep @*/ WinningNumber getWinner();
}

public class WinningNumber {
    public boolean isWinner(/*@ readonly @*/ LotteryTicket ticket );
}

public class Person {
    /*@ peer @*/ LotteryTicket ticket ;
    /*@ peer @*/ Lottery lottery ;
    public void play(/*@ rep @*/ Lottery lottery );
}
```

The lottery and the person objects are created from the main method. That is why they are peer to each other. Because both perform a write operation on the ticket object, it has to be **peer** to them. The only write access on the WinningNumber object is from the lottery. Every other access is done through a **pure** method. Therefore, it is owned by the lottery. Method return values are annotated w.r.t the target object. That is why the return value of getWinner() is annotated with **rep**. The method isWinner is declared to be a **pure** method. Its parameters can therefore only be **readonly**. This would be found by the program anyways, since the ticket is in another context than the WinningNumber.

Memory and time consumption

This example is too small to produce representative information on memory or CPU usage. However, all steps that used no I/O-operations needed less than 10ms to run and overall, the whole algorithm used 1.952s.

4.1.3 Golf driver

Listing 4.6 contains the code of the example. The class Car has a field of type Engine and defines methods to start and stop the engine. The class Golf inherits from the class Car and defines a method to enter the golf. Finally, a GolfDriver can drive a golf. What we want to show with this example, is that the program correctly annotates the field or method signatures on the class which declared them. This was described in section 2.5.6.

Listing 4.6: Code for the golf driver

```
public class GolfDriver {
        Golf golf;
        public void driveGolf(){
                golf.enter(this);
                golf.start();
                golf.stop();
        }
        public static void main(String[] args){
                Golf mycar = new Golf();
                GolfDriver me = new GolfDriver();
                me.golf = mycar;
                me.driveGolf();
                System.out.println("finished␣driving");
        }
}

public class Car {
        Engine engine;
        public Car(){
                engine = new Engine();
                engine.isRunning = false;
        }
        public void start(){
                engine.isRunning = true;
        }
        public void stop(){
                engine.isRunning = false;
        }
}

public class Engine {
        boolean isRunning;
}

public class Golf extends Car {
        GolfDriver driver;
        public void enter( GolfDriver  driver ){
                this . driver  = driver ;
        }
}
```

Listing 4.7 shows the annotated signature found by the type inferring program.

Listing 4.7: Annotated signature of the golf driver

```
public class GolfDriver {
        /*@ peer @*/ Golf golf ;
}

public class Car {
        /*@ rep @*/Engine engine;
}

public class Golf extends Car {
        /*@ peer @*/ GolfDriver  driver ;
        public void enter(/*@ peer @*/ GolfDriver  driver ){...}
}
```

Again, the example is too small to provide representative information on memory and time consumption. Please see the other examples for more information on this subject.

4.1.4 General information on memory consumption

To be able to estimate the memory consumption it is necessary to know the memory usage of the objects of the most important classes. If a program is known to generate n objects then the type inferring program will need $n * 40 bytes$ memory to store the objects. For each object at least one WReference is inserted. Otherwise, it may be difficult to get an approximation of how many WReferences will be inserted in the Extended Object Graph. This would require an estimation on how many write accesses between *different* objects occur.

Class	*Size of 1 Object*
GObject	40 bytes
VarReference	24 bytes
WReference	32 bytes

Each time a method is invoked with different parameters that are of reference type, a VarReference will be inserted for each parameter. If the number of method parameters can roughly be estimated, this can help to get an approximation of how many VarReferences will be inserted in the Extended Object Graph during the type inference.

4.2 Related work

We already mentioned the semester project of Macro Meyer [15]. Its goal is to incorporate the annotations found by the program of this project into the analysed Java program. Additionally, a viewer or editor is developed to visualise all steps of the algorithm.

At the moment of this writing Nathalie Kelleberger is working on her master project [20]. The goal of her project is static Universe type inference.

Also mentioned above is the master project of Alisdair Wren [9]. In this project the Extended Object Graph is formally introduced. The process to infer the Ownership types is only described to the point of structuring the object store. The mapping of this dynamic structure of the program to its static structure is only shortly outlined. Furthermore, no prove of concept is provided in terms of implementation of a prototype. For this reason, the information gathering part of the process is not covered at all.

4.3 Future work

4.3.1 Arrays

Due to limitations of the JVMTI, arrays were not supported in this project. This problem could be resolved by using the bytecode instrumentation features provided by JVMTI.

As stated in section 1.2.3 on page 16, arrays need two type modifiers to be annotated. The first describes the relationship between the **this** object that references the array and the array object itself. The second describes the context of the objects referenced by the array fields relative to the array object.

We suggest to use two different variables to represent these two types. Nothing special has to be done for the first. This is simply the normal field, parameter or return value declared in the class. It is recognised as variable by the type inferring program as it is. Probably a new variable type has to be implemented, that links the first variable to the second. When annotating the array variables, this would be much more convenient to find both types.

The second variable needs more explanation. It stands for the array fields. If the handling for object fields would be followed, a variable for each array field would have to be inserted. There are several reasons why this should not be done. From the point of view of type inference it is not relevant which reference was stored in which array field. All that is important is *which* reference was stored in *an arbitrary* array field. So, no information is lost if all the correspondent variable references have the same variable stored. Another reason is that all array fields must have the same type. If a variable for each field would be used, an additional level of harmonisation would be necessary. By using only one variable, the normal harmonisation step can be performed and all array fields will get the same type.

As stated in section 1.2.3 the second type modifier can never be **rep**. This property has to be ensured. Giving that the array object will never perform a write operation on an object referenced by it, never a write reference will be inserted in the Extended Object Graph. This excludes the array object from becoming the direct dominator of an object referenced by one of its fields. Thus, this constraint will already be met by the algorithm as it is.

Also stated in section 1.2.3 is that multidimensional arrays are also annotated with only two type modifiers and that the arrays, that form the multidimensional array, have to be in the same context. We suggest to ensure this property as follows. When a one dimensional array aone is added to a multidimensional array amult, the write references aone—amult and amult—aone are inserted in the Extended Object Graph. By creating this cycle, the conflict resolution step will make both objects peer to each other. Since both array objects are peer to each other the references from amult to the one dimensional arrays is not annotated. Therefore, no variable references are inserted between them. This situation is illustrated in listing 4.8 and figure 4.2.

Listing 4.8: Two dimensional array and the corresponding one dimensional arrays.

```
Object [][]  amult = new Object[2][15]; // will  be annotated rep  readonly
Object []  aone = new Object[15];
Object []  a2 = new Object[15];
 ...
aone[7]  = obj1;
a2[3]  = obj2;
amult[0]  = aone;
amult[1]  = a2;
```

Another requirement is that all fields of different one dimensional arrays that form a multidimensional array have to be of the same type. We suggest, that for each one dimensional array that is added to a multidimensional array, its variable is registered with the one from the multidimensional array. When the algorithm comes to the harmonisation and annotation step, all variables from all the one dimensional arrays—aone[i] and a2[i] in figure 4.2—are taken together and harmonised in one step.

There is one problem that we do not know yet, how it should be resolved. When an object performs a write operation on an object that is referenced by an array field, this means that the neither type modifier of the array can be **readonly**. The same problem occurs as for all other variables. We do not know how to identify, which variable was used

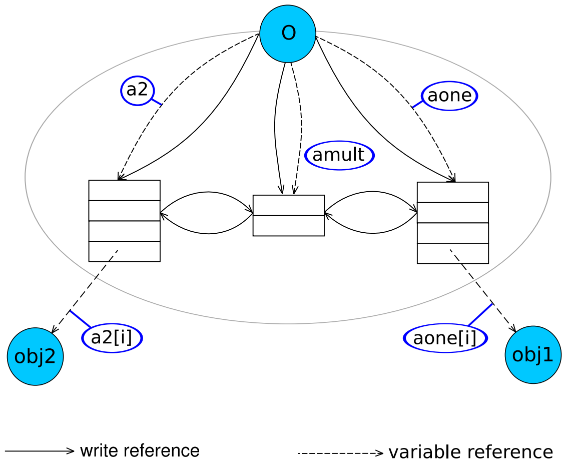

Figure 4.2: The Extended Object Graph of a two dimensional array. The write references between the array objects are introduced artificially to ensure that they are in the same context. The variables aone[i] and a2[i] stand for all fields of the corresponding arrays. All variable references of the variables aone[i] and a2[i] will be harmonised together.

to access the array and can therefore not register that constraint with the variable.

The necessary information to implement our suggestion about the handling of arrays can be obtained through bytecode instrumentation. The idea is to insert method calls to a method of the tracing agent everytime an array is created or modified. That way, the missing events for arrays of the JVMTI can still be generated. We did not yet investigate all possibilities that bytecode instrumentation offers, but maybe the problem mentionned before can also be resolved in the same way.

4.3.2 Local variables

Similar to the tracing problem of arrays it is also not possible to generate events for modification of local variables in JVMTI. There may be two possibilities to solve that problem.

The first is to implement an abstract interpreter that takes the annotations found by the program of the project and performs data flow analysis on the program code. For instance, when a field is assigned to a local variable, one can deduce that the local variable must be of a super type of the type of the field. This approach would represent a combination of dynamic and static type inference.

The second solution could be, as for arrays, to use bytecode instrumentation to artificially generate the missing events. But this would introduce a very large overhead and the program tracing could become unfeasible for any usefull program.

4.3.3 Partially annotated programs

It would be very usefull, if the type inferring program could handle partially annotated programs. A programmer could then manually introduce some annotations that she knows to be necessary and correct. The program would then annotate the rest of the program by conforming to the given annotations. A warning could be issued if no correct annotations can be found based on the one given.

On another level, it would be usefull if the user could interact with the algorithm while it is running. The already mentionned editor [15] could be extended to allow the user to insert annotations directly in the Extended Object Graph while the type inferring program is still running. The program could then compute the consequences and report if a valid annotation can be found with the inserted annotations.

4.3.4 Dereferencing chains

Listing 4.9: Dereferencing chain

```
var.exp.exp. ... .f = value;

exp := [ field |method]
```

To check if a write access after a dereferencing chain is legal in the Universe type system the type rules described in section 1.2.3 are used. That means that at most one used reference can be **rep**. The rest of the references must be **peer**.

In section 2.5.1 the limitations of the information gathering where described. Because of them, we are not able to discover dereferencing chains. This means that it is not possible to check the rules above for the Universe types that the algorithm infers. This is another problem that could be attacked in future projects.

4.3.5 Further possible improvements

Of course, all aspects that where determined not to belong to the scope of the project in section 1.4 could be incorporated into the program. Following is a list of improvements that could be implemented additionally.

Pure constructors In the Universe type system it is possible to declare a constructor as **pure**. This introduces some constraints about the types of the parameters and about the objects that are potentionally created within such a constructor.

JDI Once the JDI is adapted to support the tracing of method return values, it would be much cleaner to redesign the whole program and implement it using JDI. This would eliminate the gap between the information gathering and the algorithm. One limitation for the type inference of larger programs is the size of the agent output file and the time it takes to read and parse it. This file would not be necessary with JDI. It should be quite easy to implement this change, since only the BuildUpVisitor has to be adapted.

4.4 Conclusion

The results presented in this report show, that the goal of this project was achieved. An algorithm was developed and implemented that is able to infer a correct Universe type annotation of the field and method signatures of an executable Java program. The implemented program is able to annotate programs that instantiate a few thousand objects in reasonably short time using a feasible amount of memory.

Most problems still present are rooted in technological limitations concerning information gathering. This is not only true for the language features that are not yet supported, but especially for the feasibility of the algorithm for larger real world programs. These limitations will partly be resolved through improvements of the used software. The other part can be resolved through the use of additional techniques, which employments we have already suggested.

The implemented tool has a very flexible design that allows to rapidly implement alterations of each step of the algorithm. This can be very usefull for further research on runtime type inference. Furthermore, it is very simple to implement components that observe the algorithm during its execution.

Current projects are already using the results of this project or can be used to complement them. We expect that a number of other projects in the near future will be based on these results.

Bibliography

[1] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[2] 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. Available from http://doi.acm.org/10.1145/130943.130947.

[3] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–??, 1997. Available from citeseer.ist.psu.edu/almeida97balloon.html.

[4] John Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285, New York, NY, USA, 1991. ACM Press. Available from http://doi.acm.org/10.1145/117954.117975.

[5] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 18–22 1998. ACM Press. Available from citeseer.ist.psu.edu/clarke98ownership.html.

[6] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[7] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, 2003.

[8] Werner Dietl and Peter Müller. Universes: Statically-checkable ownership for JML. *Journal of Object Technology*, 0(0):1–99, 2002. Available from http://www.jot.fm/issues/issues_Z_/.

[9] Alisdair Wren. Inferring ownership. Master's thesis, 2003.

[10] D. Clarke. Object ownership and containment, 2001. Available from citeseer.ist.psu.edu/article/clarke01object.html.

[11] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–310, New York, NY, USA, 2002. ACM Press. Available from http://doi.acm.org/10.1145/582419.582447.

[12] 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.

[13] A. Buchsbaum. A new, simpler linear-time dominators algorithm, 1998. Available from citeseer.ist.psu.edu/article/buchsbaum99new.html.

[14] Stephen Alstrup, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *DIKU technical report*, (35), 1996. Available from citeseer.ist.psu.edu/alstrup96dominators.html.

[15] Marco Meyer. Interaction with ownership graphs. http://www.sct.inf.ethz.ch/projects/student_docs/Marco_Meyer/, 2005. Semester Project.

[16] A. Potanin and J. Noble. Checking ownership and confinement properties, 2002. Available from citeseer.ist.psu.edu/potanin02checking.html.

[17] Jakarta Project. Commons digester. Available from http://jakarta.apache.org/commons/digester/.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993. Available from citeseer.ist.psu.edu/gamma93design.html.

[19] Robert Olofsson. Available from http://www.khelekore.org/jmp/.

[20] Nathalie Kelleberger. Static universe type inference. http://www.sct.inf.ethz.ch/projects/student_docs/Nathalie_Kellenberger/, 2005. Master Project.

Appendix A

Annotations XML Schema

```
<?xml version="1.0"?>

<!--
Schema for annotation files that specify Universe annotations that
should be added to existing sources.

Author: WMD

$Id: runtime_uts_inferer.tex,v 1.15 2005/07/09 14:57:20 flyners Exp $
-->

<!--
TODO:
-
-->

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">


<!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Some additional types to automatically check the input.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->

<!--
The modifiers that are valid for simple reference types.
-->
<xsd:simpleType name="SimpleUniverseModifier">
  < xsd:restriction base="xsd:string ">
    <xsd:enumeration value="peer"/>
    <xsd:enumeration value="rep"/>
    <xsd:enumeration value=”readonly”/>
  </ xsd:restriction >
</xsd:simpleType>

<!−−
The modifiers that are valid for types, including arrays.
−−>
<xsd:simpleType name=”UniverseModifier”>
  < xsd:restriction  base=”xsd:string ”>
    <xsd:enumeration value=”peer”/>
    <xsd:enumeration value=”rep”/>
    <xsd:enumeration value=”readonly”/>

    <xsd:enumeration value=”peer␣peer”/>
    <xsd:enumeration value=”peer␣readonly”/>
    <xsd:enumeration value=”rep␣peer”/>
    <xsd:enumeration value=”rep␣readonly”/>
    <xsd:enumeration value=”readonly␣peer”/>
    <xsd:enumeration value=”readonly␣readonly”/>
  </ xsd:restriction >
</xsd:simpleType>


<!−−
The modifiers that are valid for methods.
−−>
<xsd:simpleType name=”UniverseMethodModifier”>
  < xsd:restriction  base=”xsd:string ”>
    <xsd:enumeration value=””/>
    <xsd:enumeration value=”pure”/>
  </ xsd:restriction >
</xsd:simpleType>


<!−−
What target should be modified?
−−>
<xsd:simpleType name=”ToolTarget”>
  < xsd:restriction  base=”xsd:string ”>
    <!−− Modify the original Java sources −−>
    <xsd:enumeration value=”java”/>

    <!−− Create JML specification files −−>
    <xsd:enumeration value=”jml”/>
  </ xsd:restriction >
```

```
</xsd:simpleType>


<!--
With what style should the annotations be inserted?
-->
<xsd:simpleType name="ToolStyle">
  < xsd:restriction  base="xsd:string ">
    <!-- As standard type annotations, e.g. "peer␣T" -->
    <xsd:enumeration value="types"/>

    <!-- Within JML comments, e.g. "/*@␣peer␣T␣@*/" -->
    <xsd:enumeration value="jml"/>

    <!-- As escaped JML comments, e.g. "/*@␣\peer␣T␣@*/" -->
    <xsd:enumeration value="oldjml"/>
  </ xsd:restriction >
</xsd:simpleType>


<!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
The elements of our schema.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->

<!--
The top-level element consisting of one header element and
at least one class element.
-->
<xsd:element name="annotations">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="head" minOccurs="1" maxOccurs="1"/>
      <xsd:element ref="class" minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>


<!--
Some additional information at the beginning.
Should be overridable on the command line.
-->
<xsd:element name="head">
  <xsd:complexType>
    <xsd:sequence>
```

```
      <!−− Should we create a ".jml"  specification  or embed the
           annotations  in  existing  ". java"  files ? −−>
      <xsd:element name="target" type="ToolTarget"/>

      <!−− What style of Universe annotations should we use? −−>
      <xsd:element name="style" type="ToolStyle"/>

      <!−− Maybe the source of the annotations. −−>
      <xsd:element name="comment" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>


<!−−
The annotations  for  one  class .
−−>
<xsd:element name="class">
  <xsd:complexType>
    <xsd:sequence>
      <!−− Annotations for the  fields  of the  class .  −−>
      <xsd:element ref=" field "  minOccurs="0" maxOccurs="unbounded"/>

      <!−− Annotations for the methods of the  class .  −−>
      <xsd:element ref="method" minOccurs="0" maxOccurs="unbounded"/>

      <!−− Annotations for the object   initializers   .  −−>
      <xsd:element name="object_init"  type=" object_class_init  "
                   minOccurs="0" maxOccurs="unbounded"/>

      <!−− Annotations for the class   initializers   .  −−>
      <xsd:element name="class_init"  type=" object_class_init  "
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>

    <!−− The fully  qualified  name of the  class .  −−>
    < xsd:attribute  name="name" type="xsd:string" use="required"/>

    <!−− Optionally, the  relative  path to the source  file .  −−>
    < xsd:attribute  name="file"  type="xsd:string "/>
  </xsd:complexType>
</xsd:element>


<!−−
```

```
The annotation for a field .
-->
<xsd:element name="field">
  <xsd:complexType>
    <xsd:sequence>
      <!-- The annotations for the field initializer . -->
      <xsd:element ref=" field_init " minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>

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

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

    <!-- 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="UniverseModifier" default="peer"/>
  </xsd:complexType>
</xsd:element>


<!--
The annotations for a method or constructor.
-->
<xsd:element name="method">
  <xsd:complexType>
    <xsd:sequence>
      <!-- The annotation for the return type. -->
      <xsd:element ref="return" minOccurs="0" maxOccurs="1"/>

      <!-- The annotations for the parameter types. -->
      <xsd:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>

      <!-- The annotations for the local variables . -->
      <xsd:element ref="local" minOccurs="0" maxOccurs="unbounded"/>

      <!-- The annotations for object creations in this method. -->
      <xsd:element ref="new" minOccurs="0" maxOccurs="unbounded"/>
      <!-- The annotations for casts in this method. -->
      <xsd:element ref="cast" minOccurs="0" maxOccurs="unbounded"/>

      <!-- The annotations for static calls in this method. -->
      <xsd:element ref=" static_call " minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>

    <!-- The name of the method. Multiple methods can have the
         same name, the parameters resolve the overloading . -->
    < xsd:attribute name="name" type="xsd:string" use="required"/>

    <!-- 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 "/>

    <!-- Modifiers that should be added to the method.
         At the moment there is only "pure" or "". -->
    < xsd:attribute name="modifier" type="UniverseMethodModifier" default=""/>
  </xsd:complexType>
</xsd:element>


<!--
The annotations for a field initialzier .
-->
<xsd:element name="field_init ">
  <xsd:complexType>
    <xsd:sequence>
      <!-- The annotations for object creations in this initializer . -->
      <xsd:element ref="new" minOccurs="0" maxOccurs="unbounded"/>

      <!-- The annotations for casts in this initializer . -->
      <xsd:element ref="cast" minOccurs="0" maxOccurs="unbounded"/>

      <!-- The annotations for static calls in this initializer . -->
      <xsd:element ref=" static_call " minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>


<!--
The annotations for an object or class initializer .
```

```
Careful: all the initializer blocks are merged into one of each kind
for execution.
So if the annotation information comes from the runtime inference tool,
the indices might be larger than expected from one initializer alone.
-->
<xsd:complexType name="object_class_init">
    <xsd:sequence>
      <!-- The annotations for the local variables. -->
      <xsd:element ref="local" minOccurs="0" maxOccurs="unbounded"/>

      <!-- The annotations for object creations in this method. -->
      <xsd:element ref="new" minOccurs="0" maxOccurs="unbounded"/>

      <!-- The annotations for casts in this method. -->
      <xsd:element ref="cast" minOccurs="0" maxOccurs="unbounded"/>

      <!-- The annotations for static calls in this method. -->
      <xsd:element ref="static_call" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>

    <!-- The index of the initializer within the class,
          starting from zero.
    -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

    <!-- Optionally, the source line of the opening "{". -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- Modifiers that should be added to the method.
          At the moment there is only "pure" or "".
          Not supported yet, but might come...
          <xsd:attribute name="modifier" type="UniverseMethodModifier" default=""/>
    -->
</xsd:complexType>


<!--
The annotation for the return type.
-->
<xsd:element name="return">
  <xsd:complexType>
    <!-- The Java type of the return value. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>

    <!-- 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="UniverseModifier" default="peer"/>
  </xsd:complexType>
</xsd:element>


<!--
The annotation for a parameter.
-->
<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="UniverseModifier" default="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=”UniverseModifier” default=”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”>
  <xsd:complexType>
    <!−− The index of the new, starting from zero. −−>
    <xsd:attribute name=”index” type=”xsd:unsignedInt” use=”required”/>

    <!−− The Java type of the new. −−>
    <xsd:attribute name=”type” type=”xsd:string” use=”required”/>

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

    <!−− One of the Universe modifiers. −−>
    <xsd:attribute name=”modifier” type=”UniverseModifier” default=”peer”/>
  </xsd:complexType>
</xsd:element>
```

```
<!--
The annotation for a cast.
At the moment this is very limited.
The existing casts in a method are indexed, starting from zero.
No new casts can be introduced.
How could we exactly say where a new cast should be inserted??
-->
<xsd:element name="cast">
  <xsd:complexType>
    <!-- The index of the cast, starting from zero. -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

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

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

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


<!--
The annotation for a static method call.
The existing static method calls in a method are indexed, starting from zero.
-->
<xsd:element name="static_call">
  <xsd:complexType>
    <!-- The index of the static call, starting from zero. -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

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

    <!-- Optionally, the source line of the call.
          Would this really help a tool to insert the annotation?
          What if there is more than one new per line?
    -->
    <xsd:attribute name="line" type="xsd:int"/>
    <!-- One of the simple Universe modifiers, because static  calls  are
         not possible on array types.
    -->
    <xsd:attribute name="modifier" type="SimpleUniverseModifier"
           default="peer"/>
  </xsd:complexType>
</xsd:element>


</xsd:schema>
```

Appendix B

Agentoutput XML Schema

```
<?xml version=”1.0” encoding=”UTF−8”?>
<xs:schema xmlns:xs=”http://www.w3.org/2001/XMLSchema”>
  <xs:element name=”trace”>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref=”classPrepare” minOccurs=”0” maxOccurs=”unbounded”/>
        <xs:element ref=”methodentry” minOccurs=”0” maxOccurs=”unbounded”/>
        <xs:element ref=”methodexit” minOccurs=”0” maxOccurs=”unbounded”/>
        <xs:element ref=” fieldmodification ” minOccurs=”0” maxOccurs=”unbounded”/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
<!−−XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX−−>
<!−− class prepare event −−>
  <xs:element name=”classPrepare”>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref=” field ” minOccurs=”0” maxOccurs=”unbounded” />
        <xs:element ref=”method” minOccurs=”0” maxOccurs=”unbounded” />
      </xs:sequence>
      < xs:attribute name=”name” type=”xs:string”/>
      < xs:attribute name=”id” type=”xs:int” />
    </xs:complexType>
  </xs:element>
<!−− field −−>
  <xs:element name=”field”>
    <xs:complexType>
      < xs:attribute name=”id” type=”xs:int” use=”required” />
      < xs:attribute name=”name” type=”xs:string”/>
    </xs:complexType>
  </xs:element>
<!−− method −−>
  <xs:element name=”method”>
    <xs:complexType>
      < xs:attribute  name=”id” type=”xs:int”  use=”required”  />
      < xs:attribute  name=”name” type=”xs:string”/>
      < xs:attribute  name=”signature” type=” xs:string ”/>
    </xs:complexType>
  </xs:element>
<!−−XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX−−>
<!−− method entry event −−>
  <xs:element name=”methodentry”>
    <xs:complexType>
      <xs:sequence>
        <xs:element  ref=”method” minOccurs=”1” maxOccurs=”1” />
        <xs:element  ref=”callerObj ”  minOccurs=”1” maxOccurs=”1” />
        <xs:element  ref=”targetObj”  minOccurs=”1” maxOccurs=”1” />
        <xs:element  ref=”param” minOccurs=”0” maxOccurs=”unbounded” />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
<!−− callerObj −−>
  <xs:element name=”callerObj”>
    <xs:complexType>
      < xs:attribute  name=”id” type=”xs:int”  use=”required”  />
    </xs:complexType>
  </xs:element>
<!−− targetObj −−>
  <xs:element name=”targetObj”>
    <xs:complexType>
      < xs:attribute  name=”id” type=”xs:int”  use=”required”  />
    </xs:complexType>
  </xs:element>
<!−− param −−>
  <xs:element name=”param”>
    <xs:complexType>
      < xs:attribute  name=”id” type=”xs:int”  use=”required”  />
      < xs:attribute  name=”value” type=”xs:int”  use=”required”  />
    </xs:complexType>
  </xs:element>
<!−−XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX−−>
<!−− method exit event −−>
  <xs:element name=”methodexit”>
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element ref=”method” minOccurs=”1” maxOccurs=”1” />
        <xs:element ref=”targetObj” minOccurs=”1” maxOccurs=”1” />
        <xs:element ref=”returnValue” minOccurs=”1” maxOccurs=”1” />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
<!−− returnValue −−>
  <xs:element name=”returnValue”>
    <xs:complexType>
      < xs:attribute name=”id” type=”xs:int ” use=”required ” />
    </xs:complexType>
  </xs:element>
<!−−XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX−−>
<!−− field modification event −−>
  <xs:element name=”fieldmodification”>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref=”field ” minOccurs=”1” maxOccurs=”1” />
        <xs:element ref=”modifiedObj” minOccurs=”1” maxOccurs=”1” />
        <xs:element ref=”modifyingObj” minOccurs=”1” maxOccurs=”1” />
        <xs:element ref=”referencedObj” minOccurs=”1” maxOccurs=”1” />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
<!−− modifiedObj −−>
  <xs:element name=”modifiedObj”>
    <xs:complexType>
      < xs:attribute name=”id” type=”xs:int ” use=”required ” />
    </xs:complexType>
  </xs:element>
<!−− modifyingObj −−>
  <xs:element name=”modifyingObj”>
    <xs:complexType>
      < xs:attribute name=”id” type=”xs:int ” use=”required ” />
    </xs:complexType>
  </xs:element>
<!−− referencedObj −−>
  <xs:element name=”referencedObj”>
    <xs:complexType>
      < xs:attribute name=”id” type=”xs:int ” use=”required ” />
    </xs:complexType>
  </xs:element>
<!−−XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX−−>
</xs:schema>
```

Appendix C

Configuration XML Schema

```
<?xml version=”1.0” encoding=”UTF−8”?>
<xs:schema xmlns:xs=”http://www.w3.org/2001/XMLSchema”>
  <xs:element name=”configuration”>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref=”agentoutput” minOccurs=”1” maxOccurs=”1”/>
        <xs:element ref=”puremethods” minOccurs=”0” maxOccurs=”1”/>
        <xs:element ref=”annotation” minOccurs=”1” maxOccurs=”1”/>
        <xs:element ref=”observers” minOccurs=”0” maxOccurs=”1”/>
        <xs:element ref=”algorithm” minOccurs=”1” maxOccurs=”1”/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

<!−− the file into which the agent has written its output −−>
  <xs:element name=”agentoutput”>
    <xs:complexType>
      < xs:attribute name=”file” type=”xs:string ” use=”required” />
    </xs:complexType>
  </xs:element>

<!−− a file, which conforms to the annotations.xsd, that specifies
              which methods are pure in the program under consideration −−>
  <xs:element name=”puremethods”>
    <xs:complexType>
      < xs:attribute name=”file” type=”xs:string ” use=”required” />
    </xs:complexType>
  </xs:element>

<!−− the file, into which the infered annotations will be written to −−>
```

```
  <xs:element name="annotation">
    <xs:complexType>
      < xs:attribute  name="file"  type=" xs:string "  use="required"  />
    </xs:complexType>
  </xs:element>

<!-- the list of observers that have to be instantiated by the tool -->
  <xs:element name="observers">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="observer" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

<!-- defines the classname of an observer to be instantiated -->
  <xs:element name="observer">
    <xs:complexType>
      < xs:attribute  name="classname" type="xs:string"  use="required"  />
    </xs:complexType>
  </xs:element>

<!-- the list of GraphVisitors that defined the algorithm and
     have to be instantiated by the tool -->
  <xs:element name="algorithm">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref=" visitor " minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

<!-- defines the classname of a GraphVisitor to be instantiated -->
  <xs:element name="visitor">
    <xs:complexType>
      < xs:attribute  name="classname" type="xs:string"  use="required"  />
    </xs:complexType>
  </xs:element>


</xs:schema>
```