VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

The Fox — A Tool for Java Object Graph Analysis

Alex Potanin

Technical Report CS-TR-02/28
November 2002

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

The Fox — A Tool for Java Object Graph Analysis

Alex Potanin

Technical Report CS-TR-02/28
November 2002

Abstract

Examination of the memory graphs of object inter-dependencies is hard. Current methods are either too time consuming for the average software developer or they ignore important information. I present a flexible query language to analyse a memory graph of a given Java program. I describe a tool, called Fox, that employs the query language to work with heap snapshots of running Java programs, and present a selection of interesting metrics collected using this tool. The Fox tool and the query language it supports will allow the researchers in the area of aliasing in object-oriented systems to verify their theories quickly and reliably.

Publishing Information

This is a VUW Honours Report.

Author Information

Alex Potanin was a BSc Honours student in 2002 under the supervision of James Noble and Robert Biddle.

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

The Fox — A Tool for Java Object Graph Analysis

Alex Potanin

Supervisors: James Noble and Robert Biddle

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

Examination of the memory graphs of object inter-dependencies is hard. Current methods are either too time consuming for the average software developer or they ignore important information. I present a flexible query language to analyse a memory graph of a given Java program. I describe a tool, called Fox, that employs the query language to work with heap snapshots of running Java programs, and present a selection of interesting metrics collected using this tool. The Fox tool and the query language it supports will allow the researchers in the area of aliasing in object-oriented systems to verify their theories quickly and reliably.

Contents

List of Figures

List of Tables

Acknowledgments

Firstly, I would like to thank my family, who have been great in helping me through this year. I would like to thank my supervisors: James Noble and Robert Biddle for their constructive feedback and help. I am also grateful for various suggestions expressed by people in the area of aliasing. Finally, I am happy to have been surrounded by my fellow graduate software engineering students: Dan, Rilla, Pippin, Stuart, Daniel, Angela, Mike, Matt, Kirk, and Craig who helped me with their suggestions throughout the year.

I would also like to thank the Freemasons for granting me their scholarship for this year, and Victoria University of Wellington for awarding me with VUW Graduate Award. Portions of this work are supported by the Royal Society of New Zealand Marsden Fund.

Chapter 1

Introduction

As computers grew in computational power, so did the complexity of the software that came with them. To help manage the growing size of software development projects, researchers developed the concept of *object-oriented programming*. As one of the popular books on the subject defines it [2]:

> Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

Object-oriented programming has proved to be extremely popular and has been widely adopted by the programming community. It has been shown to improve the understandability, maintainability, and reusability of the software artifacts [26]. Programs are now structured as collections of classes that model things in the real world, and program execution amounts to a large number of instances of these classes (*objects*) working together to achieve a common goal.

One of the characteristics of object-oriented programs is that there are a large number of objects present inside the program's memory during its execution. Objects are connected to each other via *references* and objects utilise these references to issue commands or send messages to other objects with which they cooperate. Thus, the contents of a typical program's memory can be thought of as a directed graph of objects, known as an *object graph*.

Software engineering, among many things, is concerned with software reliability. Given that the object graph lies at the foundation of most modern software products, understanding its behaviour and guaranteeing its reliable operation forms one of the essential tasks for researchers.

The study of inter-object relationships in object-oriented systems requires the analysis of large data structures corresponding to the object graphs. The Java programs we looked at while working on this project contained between 3,000 and 350,000 objects. The complexity of these object graphs is typically unexamined by the programmer but that does not mean that it is free of errors and other defects. To be able to examine such large systems, researchers and programmers alike use tools that allow object-to-object traversals and visualisation of various parts of the graph [6, 9].

One of the common ways to analyse this structure (for example to gather statistical data such as average number of incoming references, number of nodes in the object graph etc.) is to write a specific program that will analyse the graph [19]. This project presents a tool called the Fox Query-Based Debugger that allows systematic querying and examination of the results related to the object graph.

The major motivation behind the development of Fox was the lack of tools that allow a detailed examination of object graphs, in particular the study of encapsulation and ownership. Furthermore, we wanted a generic tool that can be extended further with the least amount of effort. Fox supports a query language that is subject to further development making it possible to maximise the reusability of Fox's measurement abilities.

Fox is oriented towards the analysis of the static snapshots of memory of a running Java program. The major data structures that are subjected to querying are the complete object graph and the ownership tree constructed from it [9].

This report is structured as follows:

Chapter 2, **Background**, introduces the essential concepts involved in the study of object graphs and memory structures. It covers the ideas of aliasing, uniqueness, encapsulation, ownership, and class confinement. It then goes on to introduce different approaches to the study of object graphs, covering related tools in visualisation, control flow based analysis, and query-based debugging.

Chapter 3, **Fox Project Overview**, explains where my work fits in the study of aliasing and what I tried to achieve with my project.

Chapter 4, **The Fox Query Language**, describes the Fox Query Language (FQL[1]) that forms the foundation of Fox. A user is asked to express his or her requests in this language that Fox accepts, and produces required results. To help illustrate the query language, a number of simple examples is presented.

Chapter 5, **The Fox Query-Based Debugger**, introduces the tool developed as part of this Honours Project. It goes into the history of the tool development, covers in detail the design and the rationale behind it, and presents how this tool fits into the world of object graph exploration.

Chapter 6, **Object Graph Exploration Using Fox**, describes a number of experiments that were conducted directly utilising the tool. It shows a number of contributions made by our tool in the form of the measurements in the area of ownership, confinement, general object graph properties, aliasing and more.

Finally, chapter 7, **Conclusion**, concludes this report, once more outlining the contributions made by my Honours Project. It also includes a section on future work, presenting a number of possible extensions to the tool and the query language, mostly driven by the requirements laid out by the studies that the existence of Fox allows us to conduct.

[1] Pronounced "Ef-Qu-El".

Chapter 2

Background

In this chapter, we present a number of fundamental concepts used in object-oriented programming and object graphs. We will examine in detail object graphs and the idea of aliasing. We will then describe some areas of study related to aliasing, which include uniqueness, encapsulation, ownership, and confinement, and survey a selection of tools related to object graph analysis.

2.1 Object Graphs

An object graph, the object instances in the program and the links between them, is the skeleton of an object-oriented program. Because each node in the graph represents an object, the graph grows and changes as the program runs: it contains just a few objects when the program is started, gains more objects as they are created, and loses objects when they are no longer required. The structure of the graph (the links between objects) changes too, as every assignment statement to an object's field makes or changes an edge in the graph.

Figure 2.1 illustrates the object graph of a simple part of a program, in this case, a doubly linked list of `Student` objects. The list itself is represented by the `LinkedList` object (an instance of the `LinkedList` class, presumably), which has two references to `Link` objects representing the head and tail of the list. Each `Link` object has two references to other `Link` objects, the previous and the next `Links` in the list, and a third reference to one of the `Student` objects contained in the list. Although the overall structure is clearly a general directed graph with many cycles, rather than a tree or a directed acyclic graph, some objects (such as the `Student` "Alice") are accessed uniquely by only a single reference.

Object graphs form the foundation of any object-oriented program. They represent the model of the world created by the software designer when describing the system of classes. The methods employed by modern object-oriented design (classes, associations, interfaces, inheritance, packages, patterns, UML, CRC Cards, etc.) are ultimately techniques for defining object graphs by describing the contents of the objects and the structure of the links between them.

Part of memory where the information about all objects in the object graph is stored is usually referred to as the *heap*. In Java, the heap is the only place where an object can be stored; in C++, for example, objects can also be allocated in the program's stack. When we examine Java programs, we look at their heap, which gives us information about every single object.

Inside the memory of a program, every single object has a 32-bit ID assigned to it. There is no guarantee whether this ID is going to be reused after the object is destroyed but it allows us to identify a single object uniquely among the other objects in program's memory, throughout that object's lifetime.

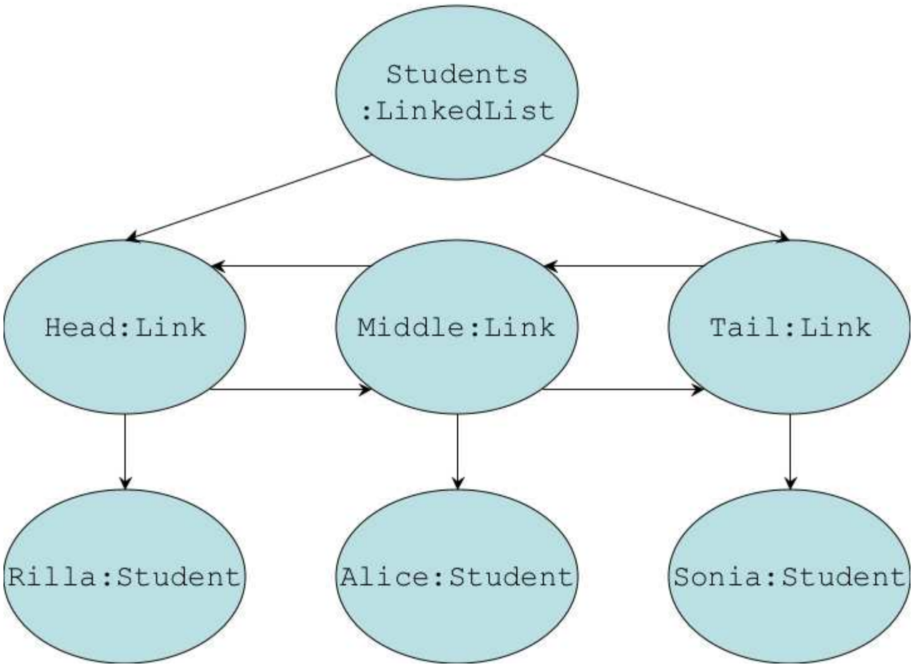


Figure 2.1: An object graph of a doubly linked list

2.2 Aliasing

Aliasing occurs when there is more than one pointer referring to one object. This causes the state of the object with respect to its referrers to be compromised because one referrer can change the state of the object without others knowing about it.

In popular programming languages aliasing is endemic and unavoidable, as even something as simple as the assignment statement causes an extra alias to be created. There has been much research done that addressed the problems caused by aliasing, including alias protection schemes [16, 17] and, in the case of assignment, alternatives to assignment statement in component-based engineering [8].

In this section we present a number of small examples of problems caused by aliasing. We hope that they will illustrate the motivation behind a large amount of research performed in this area.

2.2.1 A Simple Example of Aliasing

Consider the following class `Rectangle` that has fours fields: *x*, *y*, *width*, and *height*.

```
class Rectangle {
    public Integer x;
    public Integer y;
    public Integer width;
    public Integer height;

    public Rectangle(Integer x, Integer y,
                     Integer width, Integer height) {
        this.x = x;
        this.y = y;
```

```
            this.width = width;
            this.height = height;
        }
    }
```

Now, let's say we will create an instance of this class as follows:

```
    public static void main(String[] args) {
        ...
        Integer x = new Integer(100);
        Integer y = new Integer(50);
        Integer width = new Integer(300);
        Integer height = new Integer(200);
        Rectangle r = new Rectangle(x, y, width, height);
        ...
    }
```

Consider what will happen if we were to do the following later on in the `main` method:

```
    ...
    x.setValue(400);
    ...
```

This will not only change a local variable x, it will also modify the position of the top-right corner of our original `Rectangle`. This is a very simple example of the problems that can be caused by aliasing, but in the two sections below, I will present a number of examples that are closer to real life.

2.2.2 The Elements Inside a Hashtable

This example is based on the one discussed in detail by Noble, Potter, and Vitek on flexible aliasing protection [17].

Imagine a typical container like a hashtable used in a student database implementation. Figure 2.2 depicts a sample system with a `Hashtable` object having fields named *size* and *table*. Any object inside the system can use this hashtable to store information about the students. Every time a `Student` and, say, their `RawMark` are added into the hashtable, an `Entry` object is created inside the `Array` pointed at by the *table* field. The entry will store the reference to the `Student` object given to it and use it as a *key* inside the hashtable.

Now, there is no guarantee that nothing else is still pointing at the `Student` object (as shown by the arrows pointing at some of the objects out of nowhere in figure 2.2). This means that some of the objects that the hashtable relies upon for its state (i.e. the location of the elements in the array is based on their hash code) are aliased.

To clarify the state of the hashtable, consider the figure 2.3. A hashtable a points at an object with its *size* and an object with its *contents*. The elements inside the hashtable contents are objects named i, j, etc. Adding another object, say k, that is still pointed at by the external object d will cause one of the elements to be aliased and d will be potentially able to modify the state of the hashtable *contents* as shown by the arrow between d and *contents* in the figure.

This simple example illustrates that something as simple as a hashtable — widely used in many programs written in Java — has a huge potential for aliasing related errors. For example, the modification of the state of the hashtable element by some outside object can cause its hash code to change. For some implementations of the hashtable lookups, it may cause the container to lose track of the object.

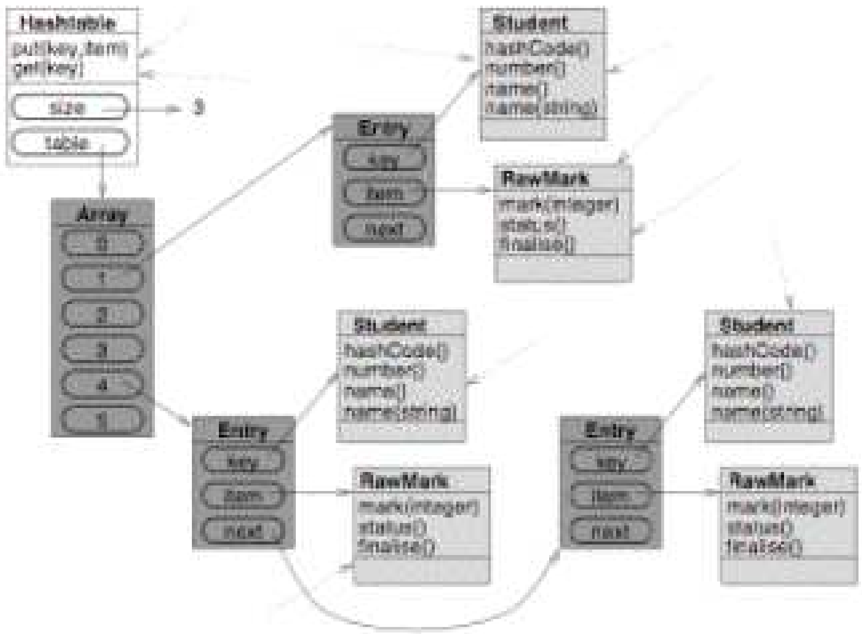


Figure 2.2: A class diagram of a typical program that utilises a hashtable (borrowed from [17])

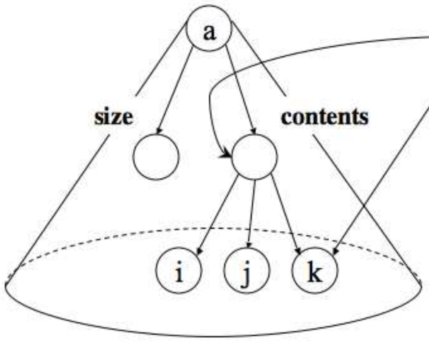


Figure 2.3: An example of aliasing

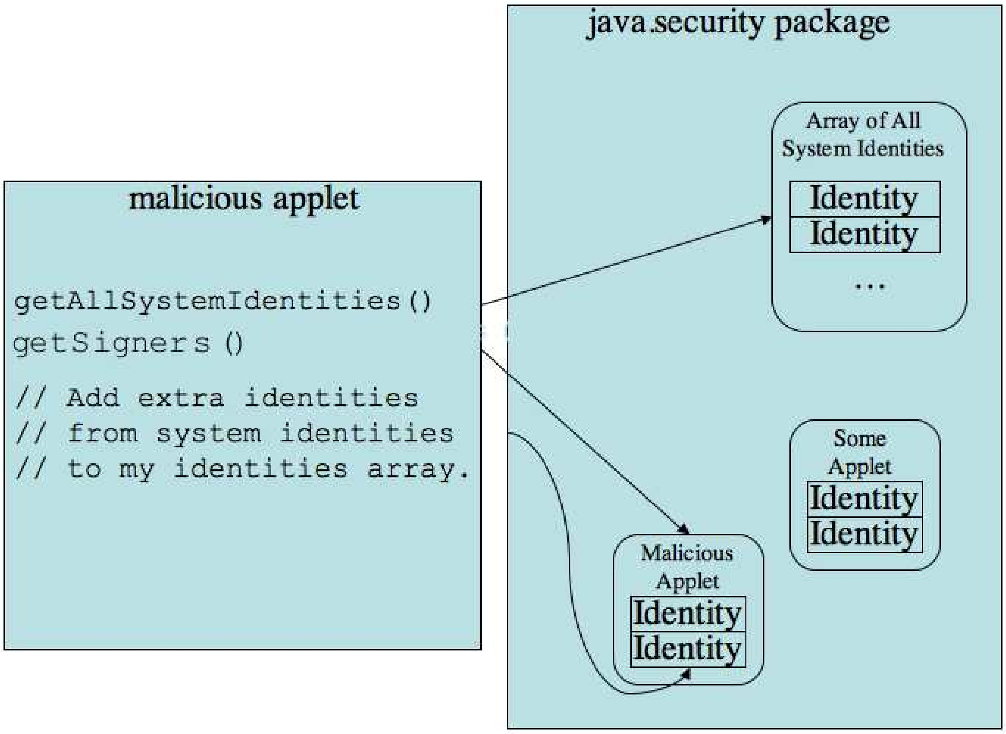


Figure 2.4: An alias to the array of identities allows a malicious applet to modify its capabilities

2.2.3 Java Applet Security Breach in JDK 1.1.1

This example is borrowed from the paper on confined types by Bokowski and Vitek [1]:

> In Java, each class object (instance of class `Class`) stores a list of signers, which contains references to objects of type `java.security.Identity`, representing the principals under whose authority the class acts. This list is used by the security architecture to determine the access rights of the class at runtime. A serious security breach was found in the JDK 1.1.1 implementation which allowed untrusted code to acquire extended access rights [23]. The breach was due to a reference to the internal list of signers leaking out of the implementation of the security package into an untrusted applet.

The way this breach could be exploited by a malicious applet is briefly outlined in figure 2.4. Every class, including the applet's own class, has an array of `Identities` stored for it inside the `java.security` package. This collections of so-called signers defines the access rights of an applet or any other class. There is also a system-wide accessible array of all possible `Identities` accessible through `java.security.IdentityScope`. Therefore, since any applet could potentially obtain the reference to its own array, it can modify it by adding the rest of the system-wide `Identities` to its own array.

The reason for this security breach lies in the code for the `getSigners` method in `java.security` package:

```
private Identity[] signers;
```

```
...
public Identity[] getSigners() {
    return signers;
}
```

As we can see, the method's implementation exposes the hidden array for modification. One of the possible fixes to this problem would have been to return a copy of the array instead:

```
private Identity[] signers;
...
public Identity[] getSigners() {
    Identity[] pub;
    pub = new Identity[signers.length];
    for (int i = 0; i < signers.length; i++) {
        pub[i] = signers[i];
    }
    return pub;
}
```

An important point to note about this example, as pointed out by Bokowski and Vitek in [1], is that "none of the standard Java protection mechanisms seem to help." Such things as "access modifiers and type abstraction are not relevant here," since "the attack does not interact with `Identity` objects, it only needs to acquire references to them and copy those references."

The paper goes on to introduce a novel and reliable way of checking for reference exposure by introducing *confined types* that ensure that all the references to the instances of classes that are confined originate from the objects in the same domain, which is taken to be a Java package. In our example above, it allows the programmers of the `java.security` package to confirm at compile time that "none of the key data structures used in code signing escape the scope of their defining package."

It should also be pointed out that confinement relates to classes, in particular their fully qualified names - it guarantees that *every* instance of a confined class is going to be referenced by instances of classes in the same package only. This is quite a strong restriction: sometimes we can have classes such that their instances are rarely referred to by instances of classes outside the defining package. We consider the latter *instantaneously* confined and we examine a number of such instances in chapter 6.

2.3 Characteristics of Aliasing

There are a number of other well-known examples of aliasing-related defects and errors. These cause concern for people who are trying to implement secure and reliable systems in reference-abundant languages such as Java. To address these issues a number of characteristics of the object graph are being explored. These include the uniqueness of objects (when they only have a single incoming reference), object encapsulation (guaranteeing that things such as hashtable elements will not be referred to from outside), and object ownership (exploring the encapsulation of objects on a scale of the whole of the object graph). In this section I will discuss these concepts in more detail.

2.3.1 Encapsulation

Encapsulation is about preserving the object boundaries. We were always taught in the software engineering courses that information that is private to the object should not be exposed. A similar concept applies to collections of objects. A private pointer to an object that constitutes the state of another object should not be shared or given to anyone. These relationships can be checked using query-based debuggers such as those by Lencevicius [14]. Our tool is also capable of doing this via the introduction of an appropriate query.

Uniqueness is the most basic way of guaranteeing encapsulation: a unique object is encapsulated within its sole referring object [3]. This implies that the surrounding object can depend upon the unique object for its private state without the aliasing-related concerns. When examining an object graph we can look at the percentage of objects that are unique versus those that are not. We can examine what classes of objects tend to stay unique when instantiated versus those that are rarely unique. All of these properties can be examined using the tool that we are presenting in this report.

2.3.2 Ownership

Every program has an underlying object graph. Objects in the graph are destroyed and created as the program runs. Java Virtual Machine (JVM) provides a service known as *garbage collection* that ensures that those objects that are no longer needed by the program are deleted so that memory that they occupy can be reused by new objects.

To be able to find unused objects, the garbage collector maintains a special selection of objects called the *root set*. The garbage, or no longer used objects are then defined as follows:

> An object is *garbage* **if and only if** there are no reference chains from some object in the root set to it.

Ownership uses the concept of root set to structure object graphs. We posit a global "fake" root r through which all the objects in the root set of an object graph can be accessed. Then, *ownership*, which is based on the notion of *dominators* from the graph theory, can be defined as follows:

> An object a *owns* another object b if all the paths from the root r to the object b go through a. In this case b is called the *owner* of a.

The implication of ownership is that no object outside the owner b is allowed to have a reference to a. Ownership allows us to structure an object graph into an implicit ownership tree.

Object ownership is in essence a generalisation of uniqueness [20] that underlies many more sophisticated alias management schemes [16, 20, 5].

To clarify this concept, consider an example in figures 2.5, 2.6, and 2.7. The first one gives a simple example of a memory graph with root R. The second one shows an ownership tree constructed from the graph just before, note how it has exactly the same number of nodes but a lot fewer edges. The third and final picture shows just the ownership tree.

2.3.3 Confinement

Every class in Java belongs to some package (if none is specified then it belongs to a so-called default package). A class is called *confined* if all instances of it are only referred to by instances of classes in the same package.

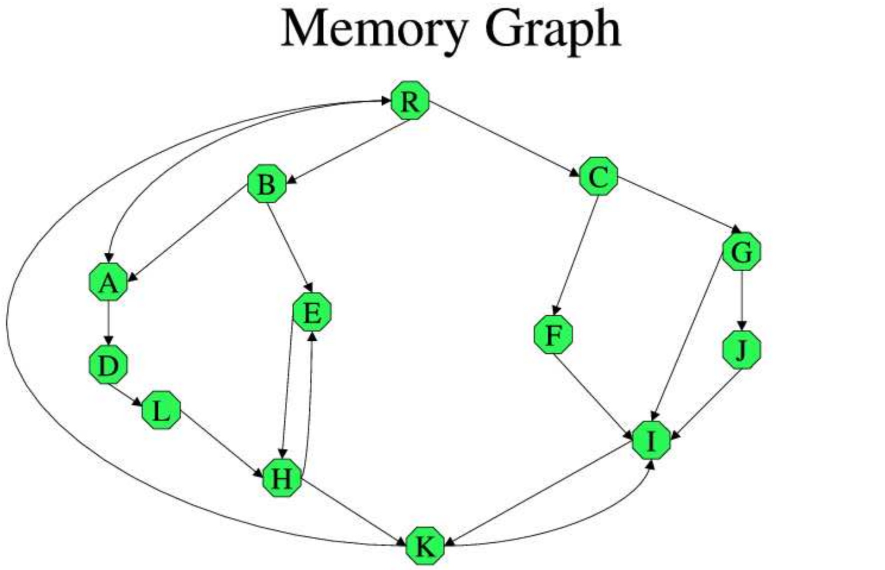


Figure 2.5: Ownership tree example (part 1 of 3): a memory graph

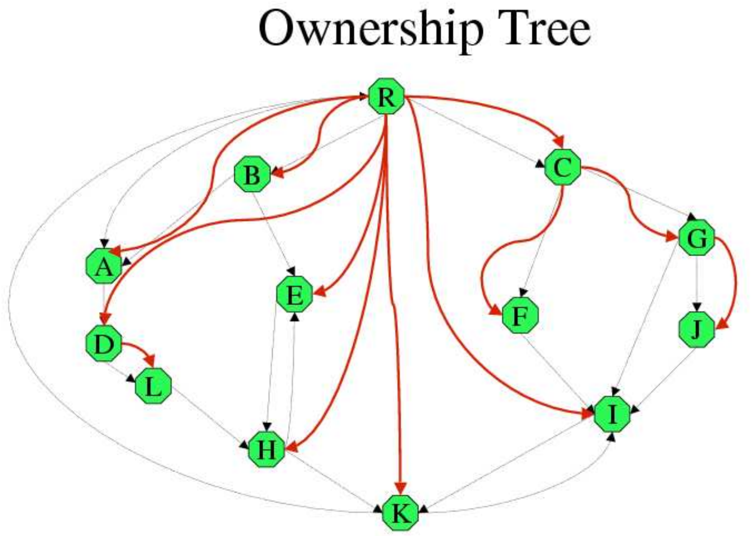


Figure 2.6: Ownership tree example (part 2 of 3): an ownership tree on top of the memory graph

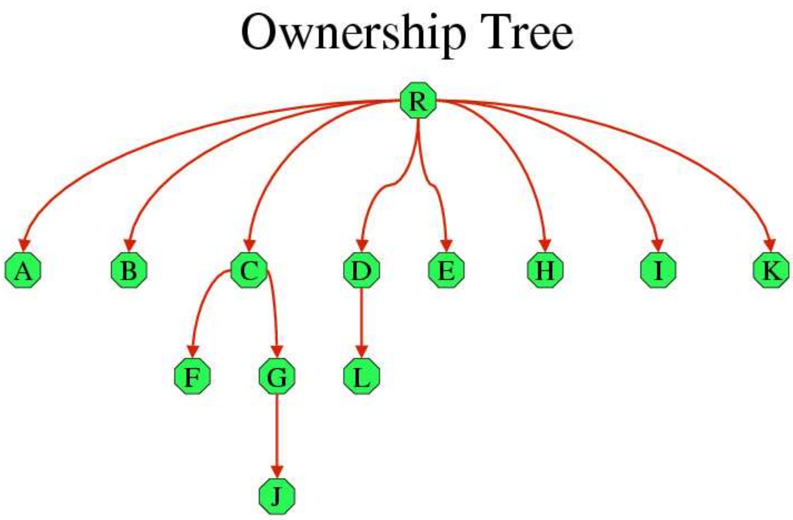


Figure 2.7: Ownership tree example (part 3 of 3): an ownership tree

The idea of confinement was explored by Bokowski and Vitek [1] and further by Grothoff, Palsberg and Vitek [7]. Confinement was viewed from the point of view of security. For example, in the JDK 1.1.1 aliasing case described above, the class `Identity` is defined inside the `java.security` package. The problem arose when instances of classes from different packages were allowed to directly access the `Identity` objects. Bokowski and Vitek in [1] demonstrate a solution to the JDK 1.1.1 aliasing problem that uses an implementation of `Identity` confined to the `java.security` package while exposing a public version of it to the public. Since inside the package a new and secure version is going to be used for storing the information, making it confined to the package renders it impossible to expose it.

The code below shows a more secure version of the solution to the JDK 1.1.1 problem that was verified using their tools that confirmed that a class `SecureIdentity` is confined to the `java.security` package.

```
confined class SecureIdentity ... {
    ...
    // the original Identity implementation
    ...
}

public class Identity {
    SecureIdentity target;
    Identity(SecureIdentity t) {target = t; }
    ... // public operations on identities;
}

private SecureIdentity[] signers;

...
```

```java
public Identity[] getSigners() {
    Identity[] pub;
    pub = new Identity[signers.length];
    for (int i = 0; i < signers.length; i++ ) {
        pub[i] = new Identity(signers[i]);
    }
    return pub;
}
```

There exists a number of tools [1, 7] that can examine code written in Java and check whether among all possible object graphs produced by the code it can ever be the case that an instance of a certain class is referred to by an instance of a class from a different package.

Finally, the package need not be the unit of confinement. For example, objects can also be confined to some general domain defined by the programmer (for example by explicitly enumerating the classes in each domain).

Both ownership and confinement relate to object's encapsulation with respect to aliasing. They restrict the amount of aliasing possible within the object graph. Ownership examines a real picture of the object graph and states which objects are within other object's "shadow" in a sense that noone outside those objects underneath the owner in the ownership tree is allowed to have references to those inside. Confinement, looks at the collection of classes which will produce object graphs when executed. From this static code, confinement checkers can derive instances of which classes will be guaranteed to stay within the "shadow" of their defining package in a sense that noone outside those objects whose classes are in the same package will have references to them throughout all possible lifetimes of a program.

2.4 Object Graph Analysis

Analysing object graphs has always been a popular topic in object-oriented software engineering research [29, 7, 20, 11, 14, 9]. One of the motivations is that they can provide a valuable insight into the real behaviour of a given program, sometimes different from the intended behaviour created by the programmer(s) who wrote the underlying classes. Such analysis can help us with understanding, debugging, and maintaining object-oriented programs.

There are many approaches taken to help with the understanding of object graphs. They include dynamic visualisation of the inter-object relationships as the program runs [29, 9], code analysis predicting all possible object graphs resulting from the underlying class structure [7], and query-based debuggers that verify that certain inter-object relationships hold throughout the graph existence [14].

The study of memory structures is a nontrivial task. Java programs that we have analysed contained anywhere between 3,000 and over 400,000 objects at the time that a memory snapshot was taken. Thus people usually tackle the object graphs in two different ways: either by examining a static snapshot of the heap in great detail, or by verifying simple, and thus fast to calculate, properties of the object structure as the program runs.

In this section we present a small selection of the tools we looked at that are designed to assist the software engineering researchers and programmers in object graph analysis.

2.4.1 Kacheck/J

Kacheck/J is presented by Grothoff, Palsberg and Vitek in [7] and is designed to analyse a large number of Java classes and derive which ones of them are confined to their defining packages and which ones are not. It is a command line tool written in Java which accepts a path to the source base of class files corresponding to the program we wish to analyse for confinement. As the result, it will report which classes are guaranteed to have no references to them from outside of their defining package throughout all possible lifetimes of the program. In addition, this tool can display which parts of the code violate confinement so that the program can be changed accordingly if necessary.

2.4.2 Lencevicius' Query-Based Debuggers

Lencevicius' work in query-based debugging is summarised in [14], it allows a programmer to either dynamically (as the program is running) or statically (as the program's memory snapshot is considered) verify relationships between objects. For example, the following query will verify if all the nodes in a linked list point to different elements:

```
LinkedListNode* l1, l2;    // Types required by the expression below.
l1.element != l2.element;  // Relationship to verify.
```

Lencevicius' work clearly demonstrates the advantage of having a flexible query language that can be used to specify the relationship that a programmer wishes to verify.

2.4.3 HOWCOME and Delta Debugging

HOWCOME is a cause-effect gap detector written by Zeller [28]. It constitutes a part of the work on the Delta Debugging Project [27].

The problem with most bugs inside programs is that the cause of an error may have happened long before the effect of an error is discovered. Zeller proposes to track the changes in the program's memory graph in the steps preceding the error being detected to recover the parts of the program relevant to the error, thus narrowing down the cause of the error without any intervention by the programmer:

> Consider the execution of a failing program as a series of program states consisting of variables and their values. Each state induces the following state, up to the failure. Which part of a program is relevant to failure? We show how *Delta Debugging* algorithm isolates the relevant variables and values by systematically narrowing the state difference between a passing run and a failing run.

2.4.4 DINO

DINO is an ownership tree visualisation tool written by Trent Hill [9]. Figure 2.8 shows the tool in action visualising how the ownership tree of a simple compiler program changes through the program's run. DINO runs in parallel to the program and detects the changes in the ownership tree, and these are visualised appropriately. Alternatively, it may also be run in the mode where it saves information about the changes in a file that can be visualised at a later stage.

2.4.5 Jinsight

Jinsight is written by IBM and allows a detailed post-mortem analysis of a Java program. It traces all object creations and destructions, method calls, memory usage and more. The results of program monitoring are stored in a file and can be visualised and analysed using a graphic analysis program that comes as part of it. Figure 2.9 shows a number of screenshots taken from its documentation [12].

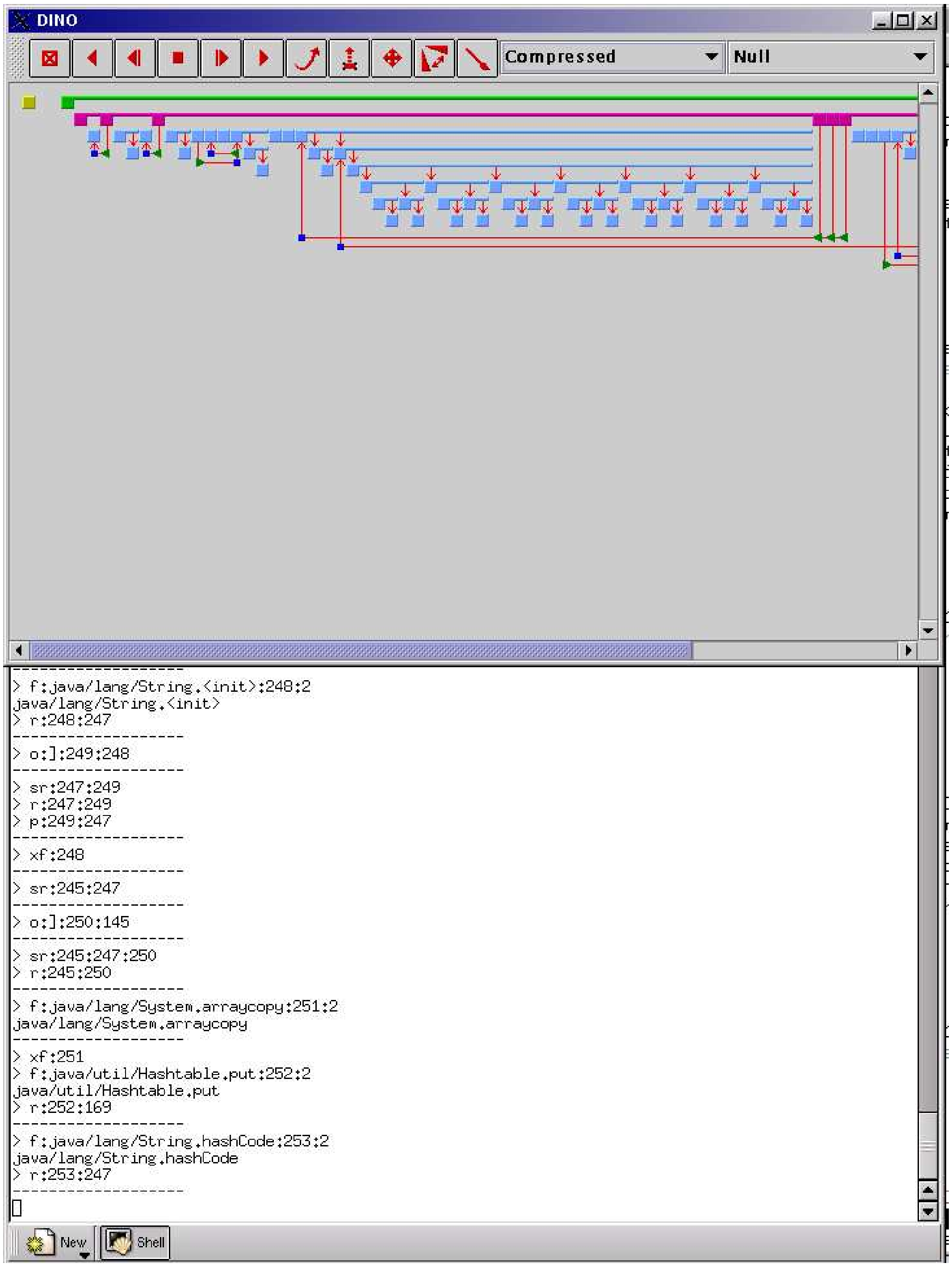


Figure 2.8: DINO — an ownership tree visualiser

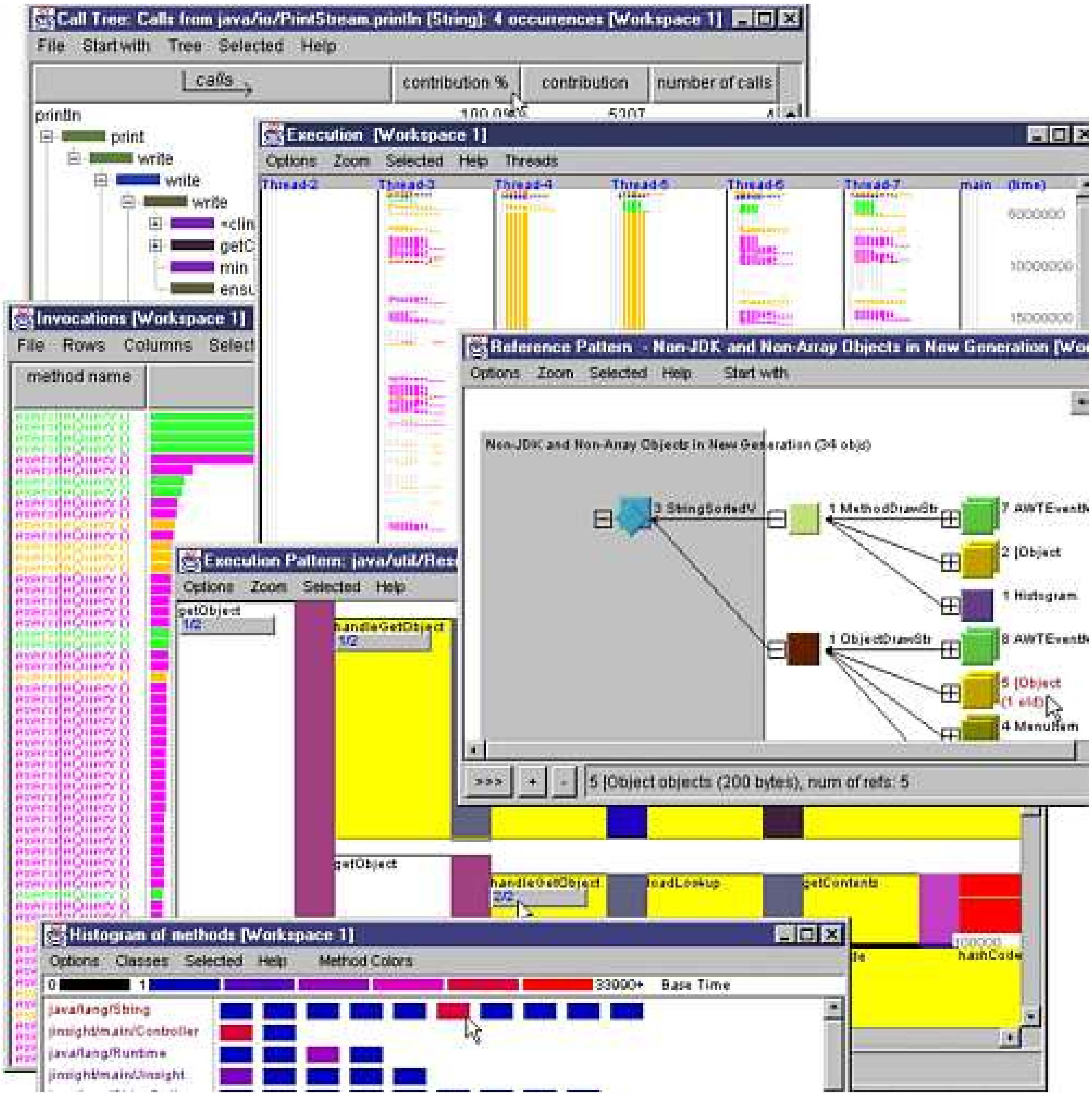


Figure 2.9: IBM Jinsight: Java memory analyser (picutre taken from [12])

Chapter 3

Fox Project Overview

This chapter explains where our work fits into the study of aliasing and what we have tried to achieve with this project.

3.1 Starting Out

We started our work by examining the current state of research in the area of aliasing, and in particular we were interested in a detailed study of ownership. For example, we found a large number of tools that would try to visualise the ownership trees inside a program, but we couldn't find a tool to estimate ownership parameters such as the average depth of an object inside a complete ownership tree, or how much effect containers such as hashtables have on ownership.

This motivated us to explore the technologies available to extract information about object graphs and ownership. Since we were trying to access information about a program's heap in the most straightforward way, we chose Java because it has a large number of standardised interfaces, and libraries to support them, that come with Sun's Java Developer Kit (JDK).

We first tried to utilise the Java Debugger Interface (JDI) that allows dynamic monitoring of any Java program as it executes. Unfortunately, if we try to track object allocations as they occur across all classes, it becomes too slow for modern computers to process and still keep the programs usable. The simplest program, with 3000 new object allocations [1], was slowed down by approximately 100 times. Thus, we decided that dynamically monitoring the ownership structure of a complete object graph was beyond our capability at the time. Visualisers successfully avoided this problem by only visualising the relevant part of the program (for example, user-written objects). We, on the other hand, needed to gather ownership information across the whole graph.

This caused us to turn our attention to trying to capture a complete image of the program's object graph examining it in detail, rather than resorting to a simplified and slow dynamic analysis.

3.2 The Heap Analysis Tool (HAT) by Bill Foote

As we were exploring the technologies at our disposal we came across the Java Virtual Machine Profiler Interface (JVMPI) that forms part of Sun's JDK 1.2 or higher [24]. Part of this interface allows the user to save a snapshot of the entire heap of any Java program into a

[1] This program is considered again in chapter 6 — nearly all of the objects in it are allocated by the Java Virtual Machine for its internal purposes.

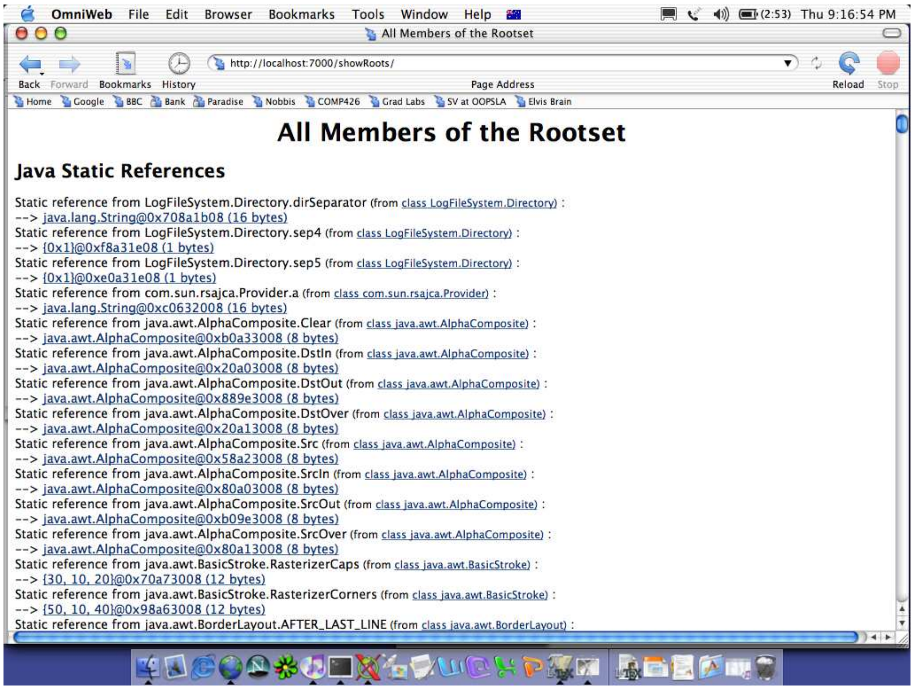


Figure 3.1: A screenshot of the HAT displaying all the objects in the root set

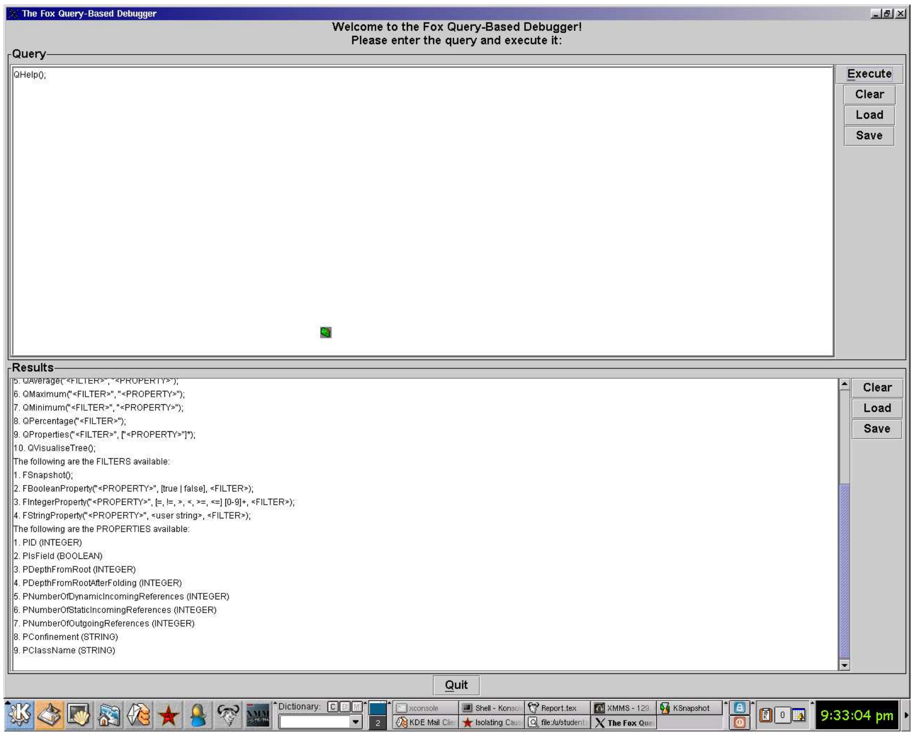


Figure 3.2: A screenshot of the Fox query-based debugger running on Solaris

file. The Heap Profiler (HPROF) library that comes as part of Sun's JDK allows the user to control when this happens, and, can produce a number of files representing heap snapshots.

These heap snapshots can be parsed using a separate Heap Analysis Tool (HAT) library written by Bill Foote [6] that creates a data structure representing the objects inside the heap being analysed. HAT then proceeds to start a server on a local machine that can be accessed using a common web browser to navigate a complete graph of objects inside the snapshot. Figure 3.1 shows a web browser displaying a page served by HAT that displays all the objects present in the root set of a given heap.

Unfortunately, HAT only allows one to browse the snapshot and has no analysis abilities.

3.3 The Rabbit That Came Out of the HAT

We proceeded to write a tool named Rabbit [19] that would use information from HAT about the objects to construct an ownership tree from the object graph, using a well known dominator construction algorithm [15]. The resulting data structures are utilised by the code in Rabbit to calculate the required metrics. Each metric calculation was specified by implementing a class conforming to the `Experiment` interface so that it could be called in order by the main function in Rabbit. Introducing a new experiment meant implementing a new class that would directly access the internal representation of the heap's objects and the ownership tree to perform its measurements.

Running a new experiment in Rabbit, even if it was a small change to the previous one, required it to be recompiled and the heap snapshot to be reloaded and all other metrics recalculated. Rabbit allowed us to perform the experiments, but was hard to use.

Rabbit did not have a query language, and we had to write a completely new class in Java that would access the data structures storing the information about the heap snapshot being analysed. Any attempt to answer a new question about the heap snapshot required the modification of the source code of the tool, complete recompilation, and reloading of the heap snapshot in question. It took a large amount of time and introduced errors in the software.

3.4 The Fox That Came After The Rabbit

After examining the related research in object graph analysis using query-based debuggers, especially the work by Lencevicius [14], we decided to improve the flexibility of our tool by introducing a query language. Figure 3.2 shows a new tool that we developed, called Fox, as a better version of Rabbit, smarter and easier to use.

Fox was used to reproduce the results obtained using Rabbit and gave us a lot of ideas for new studies. In the rest of this report, we will cover the query language, the tool itself, and a collection of studies conducted using Fox.

Chapter 4

The Fox Query Language

In this chapter we present the query language that lies at the foundation of our tool. We will go over the rationale behind its design, the language itself, and a number of small examples to illustrate its syntax.

4.1 Designing the Query Language

The study of object graphs is about objects, hence the central concept underlying the Fox Query Language (FQL [1]) is the *heap object*.

Once the information about a single heap snapshot is loaded into memory, we take a view of it as that of a single database table with each row containing information about a single object and each column denoting a particular property of the object (e.g. object's ID, object's class name etc.).

We access information about each object by accessing its *properties*. Thus, for each heap object we calculate a number of properties and store them together inside a large table so that it can be closely examined by a user.

We extend our analogy with a common database by allowing selection of objects from the table corresponding to a heap's snapshot in a manner similar to the `SELECT ... WHERE` statement in the Structured Query Language (SQL). We refer to the selection part of our query language as *filters*. Filters allow us to restrict the objects to those meeting a number of constraints on their *properties*.

Finally, to allow the user to utilise the information available to them, we provide a number of *queries* that can be run upon different selections objects returned by filters. Queries include a standard set of operations such as counting the objects or finding a minimum or a maximum value of a particular property, a set of control queries that are designed to be inserted into scripts to tell Fox when to load another heap snapshot or when to save the results, and a set of interactive queries such as `QVisualiseTree` that visualises an ownership tree of the current memory graph.

Put together, the syntax of FQL works as follows:

```
<query> :== query_name([<filter_combination>] [,query_parameters]*);

<filter_combination> :== filter_name([filter_parameters,]* <filter_combination>);
<filter_combination> :== filter_name(<filter_combination> [,<filter_combination>]*);
<filter_combination> :== FSnapshot();
```

[1] FQL is pronounced "Ef-Qu-El"

In the tables below we present precise syntax for each query and filter. Figure 4.1 gives a visual example of how a typical query works together with filters and properties. In the rest of this section, we address the three fundamental concepts of FQL: properties, filters, and queries.

4.2 Object Properties

Most information about an object can be stored inside its properties. Properties are calculated by Fox as the memory graph is loaded. Some properties such as `PID` (object's unique ID), and `PClassName` (object's class name) come from the heap itself as they are intrinsic to the objects when used by the garbage collector inside the Java Virtual Machine. These are simply read from the heap snapshot file and recorded appropriately.

Most other properties require to be calculated by Fox. While object graph specific properties such as `PNumberOfIncomingReferences` (number of objects that have a reference to the current object), `PNumberOfDynamicIncomingReferences` (number of objects that are allocated as local variables inside the methods, rather then inside the heap or as part of Java Virtual Machine, that have a reference to the current object), `PNumberOfStaticIncoming References` (number of objects that are not dynamic that have a reference to the current object), and `PNumberOfOutgoingReferences` (number of objects that the current object refers to) can be calculated by a closer examination of the memory graph, more complex properties that relate to ownership and confinement require Fox to first construct an appropriate representation of the heap information.

After the ownership tree of the graph is constructed as described by Lengauer and Tarjan [15], it is possible to calculate `PDepthFromRoot` (number of objects in the path from the root of the ownership tree to the current object, excluding the current object) for each object. Unfortunately, as we show in chapter 6, this value is usually skewed by the data structures that restrict references to their internal nodes (e.g. linked lists). To fight this, we have created a new property called `PDepthFromRootAfterFolding` that addressed the issue at hand by "folding" the reference chains consisting of objects of the same class.

Graph theoretic properties supported by Fox are `POwner` that stores the ID of an object that is a parent of the current object in the ownership tree, and `POwnershipKids` that stores a list of ID's of objects that are children of the current object in the ownership tree. For the memory graph, we store `PReferers` and `PRefersTo` that store the lists of ID's of objects that correspondingly refer to the current object or that the current object refers to.

Other properties include `PConfinement` that records whether the current object is strongly, weakly, or not instantaneously confined, and `PIsField` that is true if and only if an object is pointed at by at least one other object via that object's field.

Finally, to allow the filters to compare the property values with parameters supplied by a user, each property has a type. The types include *integer*, *boolean*, *string*, and *list of integers*. This allows the parser to check if the values supplied to the filters are compatible with those stored inside the properties.

In tables 4.1 and 4.2 I present all the properties supported at the time this was written.

4.3 Filters

For some queries, like the one to count the objects, it is more useful to work with only part of the heap snapshot, thus the FQL requires some filtering ability. We introduce the concept of *filters* to deal with it. Filters are designed to be put together so that output of one filter

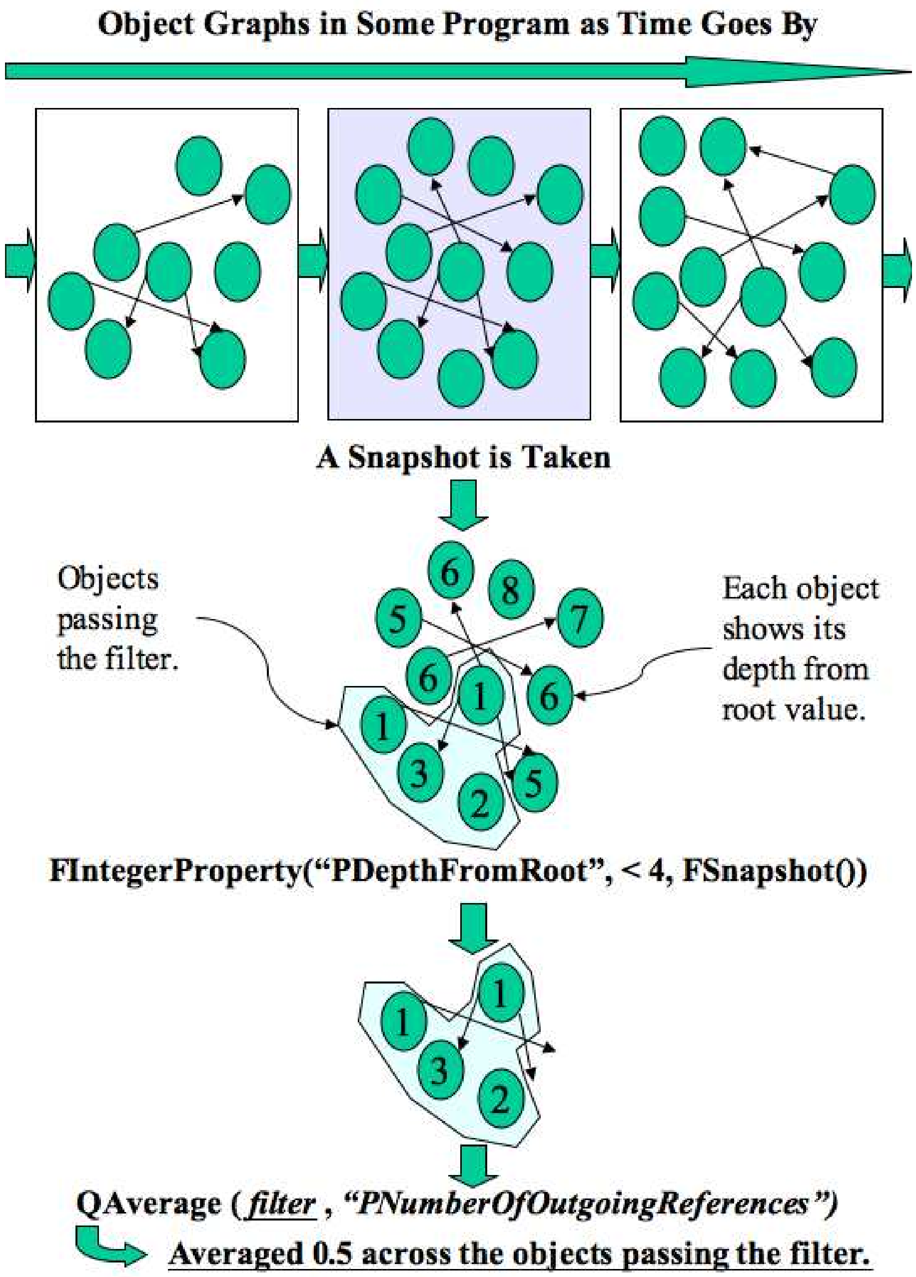


Figure 4.1: How queries, filters, and properties fit together

Property	Type	Description
PID	Integer	This property gives a unique ID of an object in the heap.
PClassName	String	This property simply stores the name of the class that the object is an instance of. It is also known as the type of the object.
PNumberOfIncomingReferences	Integer	This property gives the number of objects that point at the object in question.
PNumberOfDynamicIncomingReferences	Integer	This property gives the number of objects that point at the object in question, only it restricts them to those that have been allocated as local variables inside methods and are not used by the JVM.
PNumberOfStaticIncomingReferences	Integer	This property also gives the number of objects that point at the object in question, only it restricts them to those that are not dynamic.
PNumberOfOutgoingReferences	Integer	This property stores the number of objects that our object points at. This can be also thought of as the size of the object.
PDepthFromRoot	Integer	This property gives the depth of the object in the ownership tree constructed based upon the object graph. It can also be considered as the number of encapsulation levels above the object. (See chapter 2.)
PDepthFromRootAfterFolding	Integer	This property is similar to the `PDepthFromRoot` above, except that every time the object is owned by the object of the same class, they are considered to be at the same depth. This avoids the bias imposed by such data structures as linked lists.

Table 4.1: Properties supported by the FQL (Part 1 of 2)

Property	Type	Description
`POwner`	Integer	This property stores a direct parent of the current object in the ownership tree.
`POwnershipKids`	List of Integers	This property stores the objects that are children of the current object in the ownership tree.
`PReferers`	List of Integers	This property stores the objects that refer to the current object in the memory graph.
`PRefersTo`	List of Integers	This property stores those object that the current object refers to in the memory graph.
`PIsField`	Boolean	This property is true if and only if the object is pointed at by a field in some other object.
`PConfinement`	String	This property can take the following values: NONE, WEAK, or STRONG. It records whether an object is not confined, weakly, or strongly confined. See chapter 2 for the discussion of the notion of confinement.

Table 4.2: Properties supported by the FQL (Part 2 of 2)

serves as input to another filter. There is a special filter called `FSnapshot()` that can be used as input to other filters that returns all the objects in memory graph.

The core of the filter part of the FQL lies in `FIntegerProperty`, `FBooleanProperty`, `FStringProperty`, and `FIntegerListProperty` filters. These respectively select objects based on the restrictions to their properties, which are of the corresponding type. Additional filters include `FUnion`, `FIntersection`, and `FMinus` which accept two input filters and return objects in correspondingly union, intersection, or set difference of the sets returned by the input filters. Table 4.3 lists the filters supported by the FQL.

The reasoning behind a somewhat cumbersome syntax is the ease with which filters can be parsed and the ease with which the user can be sure that they are supplying the values of the right type. Chapter 7 discusses some ideas for future work that can improve the way filters are expressed.

4.4 Queries

Queries are where all the work is performed. There are several kinds of queries: standard queries similar to the ones that can be found in SQL, control queries that tell Fox which heap snapshot to load or where to save the results obtained so far, and a set of interactive queries designed for the user to explore the snapshot in a lot more detail. The standard queries are also expanded to return the same data as they return across the objects passing the filter, only on a per class basis, as discussed below.

Most SQL implementations allow to count the objects that meet a particular restriction, find the maximum or the minimum of some value across the objects, or to find which percentage of objects pass a given combination of filters with respect to the whole memory graph. Every query in the table accepts a combination of filters as the first parameter. `QCount` and `QPercentage` queries then proceed to count the object that pass a given filter combination. The `QAverage`, `QMaximum`, and `QMinimum` queries also accept a second parameter with the property that they base their calculations upon.

Standard queries base their calculations on all the objects returned by the filter. At times, it is more important to track the distribution of the numbers we are getting across the objects of the same class. For example, it is interesting to compare the depth in the ownership tree of objects of the type `Hashtable` and objects of the type `HashtableElement` in the hope that the latter is deeper in the tree than the former. For this purpose, we provide versions of standard queries that return the results separated for each class of objects. The result is going to be a list of class names together with the measurement across all the instances of a given class. Here is a simple example:

```
Class java.util.Hashtable has 279 instances and on
average each instance node is 3.96 deep.
Class java.util.Hashtable$Entry[]; has 297 instances
and on average each instance node is 4.90 deep.
Class java.util.Hashtable$Entry has 3990 instances
and on average each instance node is 5.37 deep.
```

Table 4.4 lists standard queries together with their per class versions.

Control queries are the kinds of queries that can be provided in the user interface. They include the commands to load a heap snapshot (`QLoadHeapSnapshot`), list currently loaded heap snapshots (`QListHeapSnapshots`), save the current contents of the window with the results of running the queries (`QSaveResults`), and clear the contents of the results window (`QClearResults`). The reason for providing this queries in addition to the user interface

<table>
<tr><th>Filter Syntax Followed by its Description</th></tr>
<tr><td>FSnapshot();

This filter represents all the objects in the heap snapshot.</td></tr>
<tr><td>FIntegerProperty("<PROPERTY>", [=, !=, >, <, >=, <=] [0-9]+, <FILTER>);

This filter selects the objects from those that pass a given filter.
If the property is an integer, it will compare it with the number given
using the comparison operator given. It will only keep those objects
that pass the comparison. If the property is not an integer, it will fail.</td></tr>
<tr><td>FBooleanProperty("<PROPERTY>", [true | false], <FILTER>);

This filter selects the objects from those that pass a given filter
and then have a given property being either true or false as given.
It will fail if the property given is not of boolean type itself.</td></tr>
<tr><td>FStringProperty("<PROPERTY>", <user string>, <FILTER>);

This filter selects the objects from those that pass a given filter.
If the property is a string, it is compared with the given user string
and if they are equal, the object is accepted. Otherwise the filter fails.</td></tr>
<tr><td>FIntegerListProperty("<PROPERTY>", [in | notin] [0-9]+, <FILTER>);

This filter selects the objects from those that pass a given filter.
If the property is an integer lists, it will see if the number given
is in the list or not in the list and will only keep those objects
that satisfy the relationship specified. If the property is not an
integer list, it will fail.</td></tr>
<tr><td>FUnion(<FILTER>, <FILTER>);

This filters selects the objects from those that pass at least
one of the filters given. The duplicates that pass both filters
are eliminated.</td></tr>
<tr><td>FIntersection(<FILTER>, <FILTER>);

This filters selects the objects from those that pass both
of the filters given.</td></tr>
<tr><td>FMinus(<FILTER>, <FILTER>);

This filters selects the objects from those that pass the first
filter but not the second one of the filters given.</td></tr>
</table>

Table 4.3: Filters supported by the FQL

Query Syntax Followed by its Description
QCount("<FILTER>"); QCountPerClass("<FILTER>"); This query simply counts and returns the number of objects passing a given filter.
QAverage("<FILTER>", "<PROPERTY>"); QAveragePerClass("<FILTER>", "<PROPERTY>"); This query accepts an integer property and calculates the average of its value across the objects passing a filter.
QMaximum("<FILTER>", "<PROPERTY>"); QMaximumPerClass("<FILTER>", "<PROPERTY>"); This query accepts an integer property and finds the maximum of its value across the objects passing a filter.
QMinimum("<FILTER>", "<PROPERTY>"); QMinimumPerClass("<FILTER>", "<PROPERTY>"); This query accepts an integer property and finds the minimum of its value across the objects passing a filter.
QPercentage("<FILTER>"); QPercentagePerClass("<FILTER>"); This query calculates what percentage of all the objects in the heap snapshot pass the filter. **NB!** In case of the per class version of this query, the percentage of objects with respect to all the objects of the same class is going to be returned.

Table 4.4: Standard queries supported by the FQL

Query Syntax Followed by its Description
`QLoadHeapSnapshot("/path/to/the/heap/snapshot.hprof");` This query loads a heap snapshot stored in a file that is pointed to by the file name parameter. The file is obtained using the HPROF library [24]. Chapter 5 gives more detail about how it works.
`QListHeapSnapshots();` This query displays the names of the loaded heap snapshots. At the time of writing only one snapshot was allowed to be loaded, but the tool can easily be extended to support more than one. Thus the query language was designed to account for it.
`QSaveResults("/path/to/the/results/file.txt", [true | false]);` This query saves the results to a file that is pointed to by the file name parameter. Second parameter specifies whether this query is allowed to overwrite the file if it already exists.
`QClearResults();` This query simply clears the window with the results. It is useful in combination with `QSaveResults` query just above.

Table 4.5: Control queries supported by the FQL

controls is to allow the user to write a reasonably long script using FQL that can be executed by Fox over an extended period of time. This script can go through a large number of heap snapshots, load each of them, process and save the results in different locations for further analysis by the user. These queries are listed in table 4.5.

The final category of queries are the interactive queries that are designed to be used when a user wants to closely examine a particular snapshot themselves, rather than letting Fox run in a batch mode. They include a help command (`QHelp`), a query to visualise an ownership tree of the memory graph (`QVisualiseTree`), and a query to traverse the memory graph of objects using HAT [6] that will serve them on a given port for the user to use any web browser to explore it (`QHAT`). These are described in table 4.6.

There is also a special query called `QProperties` that doesn't have to be used interactively. It is presented in table 4.6 and is designed to simply output a comma separated list of property values for all the objects passing a given filter. Only properties supplied as arguments will be displayed. We were using this query extensively to study distributions of various properties of objects.

4.5 Examples

These sections presents a number of simple examples that serve to illustrate the query language. They are not designed to go into the detailed analysis, rather they just provide the reader with a feel for the query language.

To count the number of objects in the heap snapshots, we need to use the `QCount` query over the objects returned by the `FSnapshot` filter:

```
QCount("FSnapshot()");
```

To get only the number of objects that have a class name `java.lang.String` we have to run the query that filters out the required objects using `PClassName` property:

Query Syntax Followed by its Description
`QHelp();` This query simply lists all the properties, filters, and queries supported by the tool. It also displays the syntax of each query and filter, and the type of each property.
`QVisualiseTree([object ID]);` This query visualises a complete ownership tree using a fish eye view [25]. If the object ID is supplied, it only shows the objects below the given one in the tree, if no object ID is supplied, the whole tree is displayed.
`QHAT([port number]);` This query will start a server using the HAT library [6] ability to allow the user to use any web browser to traverse the memory graph.
`QProperties("<FILTER>", ["<PROPERTY>"]*);` This query is used to produce a comma-separated list of property values of all the objects passing the filter. This result can be saved into a CSV file and loaded into a spreadsheet program such as MS Excel or GNUmeric for further analysis. Most of the more detailed analysis we performed was done using this query together with MS Excel.

Table 4.6: Interactive queries supported by the FQL

```
QCount("FStringProperty("PClassName", java.lang.String, FSnapshot()));
```

Finally, to figure out which percentage of objects are strings we use `QPercentage` query:

```
QPercentage("FStringProperty("PClassName", java.lang.String, FSnapshot()));
```

More extensive examples of using FQL are provided in chapter 6.

Chapter 5

The Fox Query-Based Debugger

In this section we present our tool, called Fox. Fox is designed to explore a snapshot of the program's heap taken at any time during its run. This approach allows us to look at a large corpus of Java programs and explore what really happens inside the programs memory. It provides the user with the query language that can be used to quickly extract information about an object graph.

To obtain a snapshot of any Java program, we need to run the program using special options to the `java` command. These options are supported by Sun's JDK 1.2 or higher and execute the Heap Profiler library that comes with it. The syntax of the options is as follows:

```
Hprof usage: -Xrunhprof[:help]|[<option>=<value>, ...]

Option Name and Value  Description             Default
---------------------  ----------------------  -------
heap=dump|sites|all    heap profiling          all
cpu=samples|times|old  CPU usage               off
monitor=y|n            monitor contention      n
format=a|b             ascii or binary output  a
file=<file>            write data to file      java.hprof(.txt for ascii)
net=<host>:<port>      send data over a socket write to file
depth=<size>           stack trace depth       4
cutoff=<value>         output cutoff point     0.0001
lineno=y|n             line number in traces?  y
thread=y|n             thread in traces?       n
doe=y|n                dump on exit?           y

Example: java -Xrunhprof:cpu=samples,file=log.txt,depth=3 FooClass
```

This implies that we can only get a snapshot of a Java program if we can run it from the command line. Some programs, although mostly implemented in Java, use an OS-specific binary to start them. These include some major applications as Sun's HotJava and IBM's Eclipse. Application such as these cannot be easily analysed , since HPROF cannot be used to obtain a heap snapshot.

The HAT requires the heap snapshots to be saved in a binary form, so the way we usually use HPROF is:

```
java -Xrunhprof:file=Snapshot.hprof,format=b,depth=42,thread=y,doe=n
```

Then, we would wait until the time when we want to dump the heap of the application and press `Ctrl+\` in the window where the `java` command was run. For Windows systems, it is

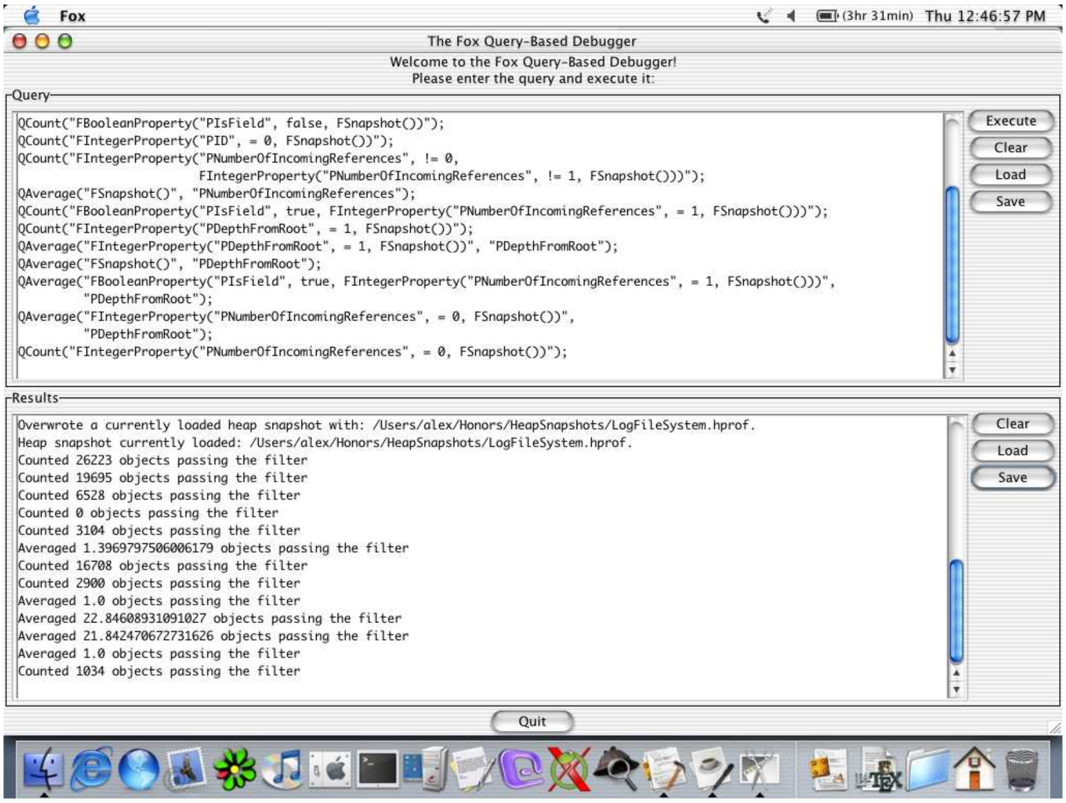


Figure 5.1: A screenshot of the Fox query-based debugger

required to press `Ctrl+Break` in the command prompt window to achieve the same result. We usually keep one heap snapshot per one file.

The control of the tool is performed via the query language discussed in chapter 4. The user interface is designed to be minimalistic with only two major text areas: one for entering and editing the queries and one for displaying the results. Each window can have its contents loaded or saved to/from a file or cleared. There are two extra buttons: to execute a query and to quit the program.

Certain tasks, such as loading the heap snapshots, are controlled from the query language and we could argue that the user interface controls should take this responsibility. The reason these functions are implemented inside the query language is to simplify batch processing of a large number of queries across a number of heap snapshots. This way, a single file with queries can be loaded into Fox and run for an extended period of time analysing different heap snapshots.

Architecture and design of Fox are explained in the section that follows. The main point of our work was to find the easiest and the most straightforward approach to study the object graphs of Java programs so that we can perform various measurements across the widest possible array of programs. Fox utilises such a method and we have so far used it successfully and plan to perform an extensive study of object graphs in the work that will follow.

5.1 Architecture

Figure 5.2 depicts the architecture of our approach to the study of memory structures. Consider a Java program being executed inside a Java Virtual Machine. We use the HPROF

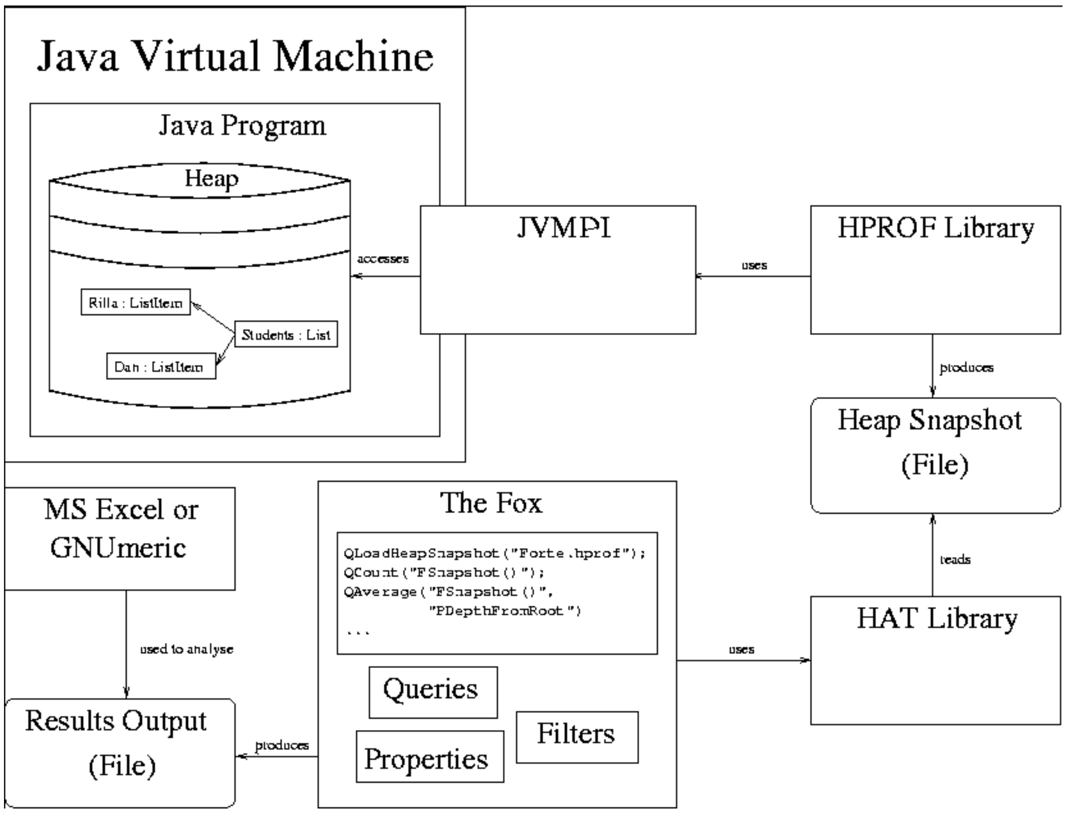


Figure 5.2: Architecture of Fox

Library that utilises the JVMPI to access the Heap of the Java program stored inside the JVM and dump a complete heap snapshot into a file.

We then start Fox that uses the HAT Library to load a file with the heap snapshot and then proceeds to construct a complete ownership tree using a fake root just above the root set. It then initialises all the heap objects using its own data structure and fills in their properties. To save the memory, Fox only leaves an array of all the heap objects together with their properties to be used by queries and filters run after the heap snapshot is loaded. The memory used up by the HAT and the ownership tree is freed by setting the respective pointers to `null` thus telling the Java garbage collector that those data structures are not required any longer. One of the experiments that we perform with Fox in chapter 6 is taking a snapshot of Fox itself and checking if those large data structures were mistakenly aliased and thus not garbage collected.

The results of running the queries upon the snapshot can be saved and further analysed using such data management programs as Microsoft Excel or GNUmeric. A particularly useful query to produce the data in appropriate format for Excel would be `QProperties()`. This outputs a comma-separated view (CSV) of the data. The latter file can be loaded into either Excel or GNUmeric.

5.2 Internal Design of Fox

Finally, to conclude this chapter's discussion of the tool, we would like to briefly point out the way we designed Fox. After having had a good experience with Rabbit, we took an approach that lead us to the simplest possible design that meets the requirement of having an easily extensible tool that handles heap data structures and allows the query language to be extended independently of them.

Figure 5.3 shows the essential classes and packages present inside Fox. There is a very little number of them and the extension of the query language is simply performed by defining new classes for queries, filters, and properties by extending the corresponding classes: `Query`, `Filter`, and `Property` with the new element being introduced. Together with the classes like `Program`, `Main`, and `Window` handling the control of the program and utility packages `tool` and `heap` the design of Fox forms a good framework into which new filters, properties and queries can be added without modifying any of the external code. As long as they follow the interface specified by the classes they extend, they will be automatically integrated into the rest of the tool.

Chapter 4 explains those queries, filters, and properties that are already implemented inside Fox. Fox will be available for download in the near future from:

`http://www.mcs.vuw.ac.nz/~alex/fox/`

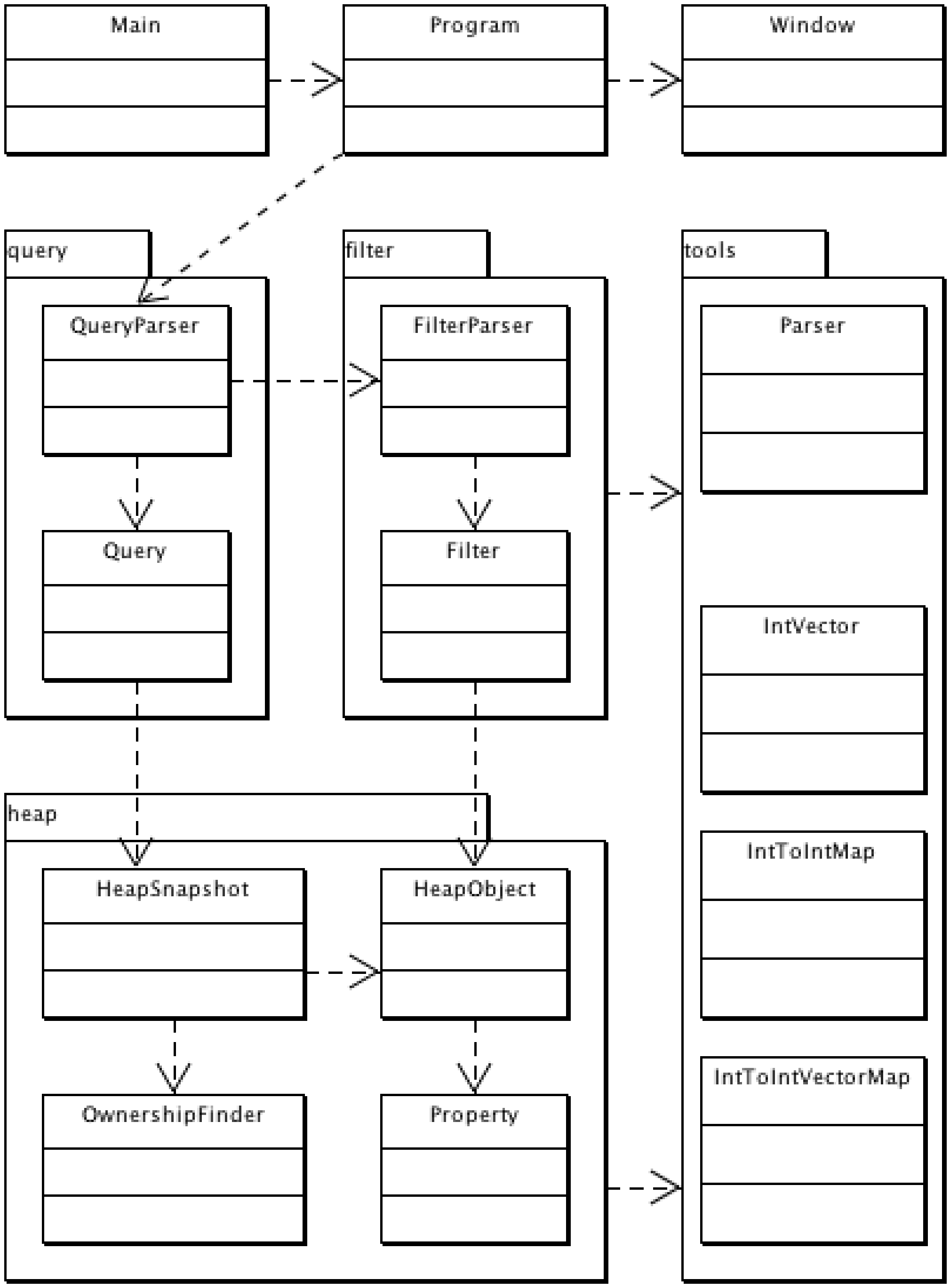


Figure 5.3: Class diagram of Fox

Chapter 6

Object Graph Exploration Using Fox

In this chapter we illustrate our tool by presenting a selection of studies performed using Fox. We concentrate on four studies: examining a single heap snapshot in detail, exploring a collection of heap snapshots taken at different stages as the program runs, presenting the distribution of incoming and outgoing references that demonstrates the presence of power laws, and finally studying a collection of programs to see how different properties of object graphs hold for a large corpus of heap snapshots, rather than a single one.

6.1 Detailed Examination of Single Heap Snapshots

This section demonstrates how Fox and FQL can be used to analyse a heap snapshots of a single program.

6.1.1 Average Ownership Depth of the Fields

In this subsection we look at the heap snapshot of the Java program called Satin that forms a part of the pen-based user interface research at University of California, Berkeley.

First, we write a query to find out the average depth of any object in the heap snapshot. We need the `QAverage` query together with the global `FSnapshot` filter and the `PDepthFromRoot` property:

```
QAverage("FSnapshot()", "PDepthFromRoot");
```

Running the query above, gives us the following result:

```
Averaged 7.83 across the objects passing the filter.
```

Which means that a typical object about 7 or 8 levels deep in the ownership tree. This number is actually biased by such data structures as linked lists (see the discussion in the section on queries). To get a more precise picture, we need to average the so-called depth after folding. The following query gives us the desired result:

```
QAverage("FSnapshot()", "PDepthFromRootAfterFolding");
```

Giving:

```
Averaged 3.44 across the objects passing the filter.
```

Now, we would like to look at the number of objects that are fields and calculate their depth after folding. We need to construct a combination of filters `FBooleanProperty` and `FSnapshot`, where the boolean property is `PIsField`. The following queries are going to answer our questions:

```
QAverage("FBooleanProperty("PIsField", true, FSnapshot())",
         "PDepthFromRoot");
QAverage("FBooleanProperty("PIsField", true, FSnapshot())",
         "PDepthFromRootAfterFolding");
```

Giving:

```
Averaged 8.39 across the objects passing the filter.
Averaged 3.11 across the objects passing the filter.
```

We can see that although 8.39 is greater than 7.83 by a whole level, if we fold up the tree, the fields are actually positioned shallower since 3.11 is less than 3.44. We can now hypothesise that while in a generic ownership tree, the fields in Satin were deeper by an extra level, possibly provided by the object containing the field, in general being a field doesn't have much effect on the object encapsulation inside other objects. It seems quite likely that fields are pointed at by unrelated objects more often than non fields. At this stage, I will stop discussing this issue as it is not supported by a large enough data set.

6.1.2 In-Degree Versus Out-Degree

This is a small study that compares the number of incoming references to each object with the corresponding number of outgoing references. The object graph in question came from a small program called the `LogFileSystem` that implements a simple log-based file system and is around 1000 lines long.

To obtain the results, the following queries were run to load the heap snapshot and produce a comma separated file plotted using MS Excel:

```
QLoadHeapSnapshot("/User/alex/Honours/HeapSnapshots/LogFileSystem.hprof");

QProperties("FSnapshot()",
            "PNumberOfDynamicIncomingReferences",
            "PNumberOfOutgoingReferences");
```

After saving the results and loading them into Excel, the graphs in figures 6.1 and 6.2 were produced. The second figure contains the part of the graph with 96.81% of all objects, but excluding the extreme-valued points.

As a small point of interest, I ran two small queries that calculated that there are 7845 objects with 1 incoming and 0 outgoing references and that there are 94 objects with 0 incoming and 1 outgoing reference. The latter surprised me and the third query was ran to verify that all the latter objects are indeed in the root set. Here are the aforementioned queries:

```
QCount("FIntegerProperty("PNumberOfDynamicIncomingReferences", = 0,
        FIntegerProperty("PNumberOfOutgoingReferences", = 1,
        FSnapshot()))");
QCount("FIntegerProperty("PNumberOfDynamicIncomingReferences", = 1,
        FIntegerProperty("PNumberOfOutgoingReferences", = 0,
```

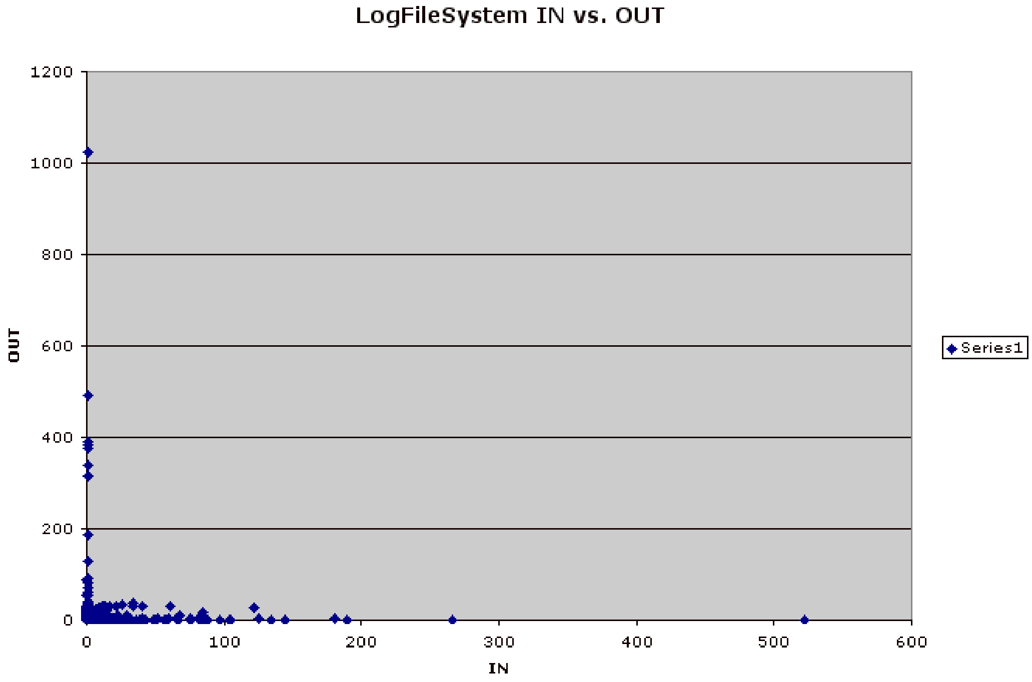


Figure 6.1: Incoming versus outgoing references in `LogFileSystem` (complete graph)

```
       FSnapshot()))");
QAverage("FIntegerProperty("PNumberOfDynamicIncomingReferences", = 0,
         FIntegerProperty("PNumberOfOutgoingReferences", = 1,
         FSnapshot()))",
        "PDepthFromRoot");
```

Giving:

```
Counted 94 objects passing the filter.
Counted 7845 objects passing the filter.
Averaged 1.0 across the objects passing the filter.
```

A more detailed analysis of incoming and outgoing references is presented in the later chapter covering the Object Graph Exploration Using Fox.

6.1.3 Visualising the Ownership Tree

Consider the following program written in Java:

```
public class HelloWorld {
    public static void main(String[] args) {
        for (int i = 0; i < 1000000; i++) {
            for (int j = 0; j < 100; j++) {
                // Do nothing.
            }
        }
    }
}
```

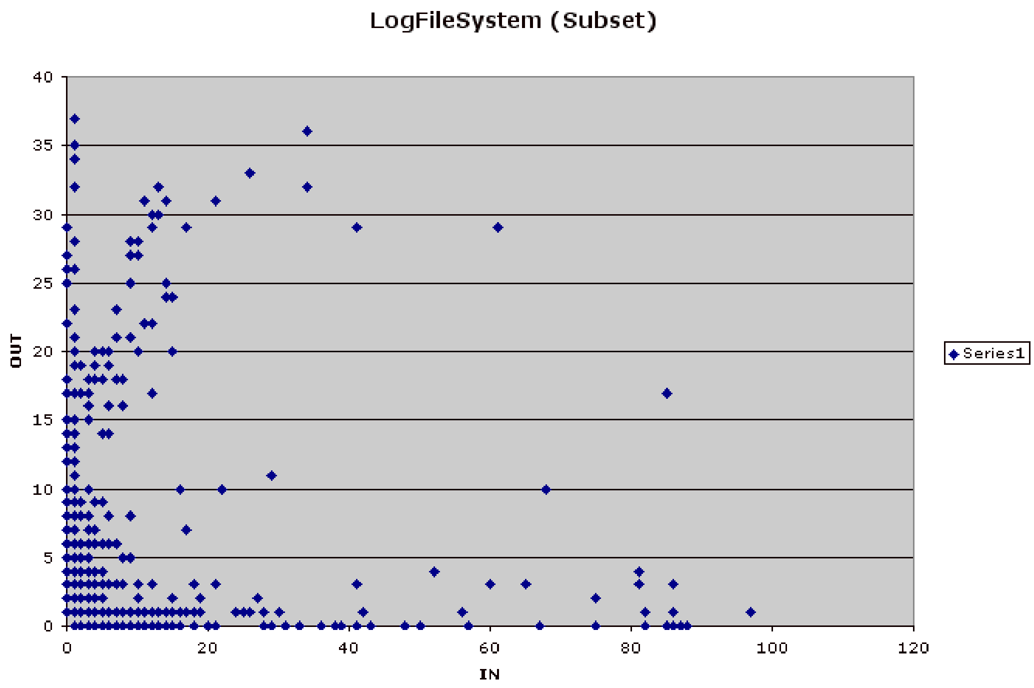


Figure 6.2: Incoming versus outgoing references in `LogFileSystem` (interesting subset of the complete graph)

From our point of view, it is a rather simple program, but surprisingly, counting the number of objects in the heap snapshot:

```
QCount("FSnapshot()");
```

Gives:

```
Counted 2633 objects passing the filter.
```

Which means that one of the simplest programs that you can write in Java will have around three thousand objects! This makes the task of visualising the ownership tree quite complicated, especially for large snapshots like Forte, containing around 350,000 objects. Running the `QVisualiseTree` query produced the fish eye view [25] shown in figure 6.3.

Fish eye view allows us to examine the objects we are interested in in greater detail, without letting other objects to get in the way. At the same time, it allows us to have an overview of the rest of the system of objects by looking at their concentration near the edges of the fish eye view.

6.2 Multiple Snapshots of a Single Program

In this section we consider several programs in detail, exploring the growth in the number of aliased objects in **ArgoUML**, the presence of power law dependency on references in the memory graphs of **Forte**, **ArgoUML**, **BlueJ**, **Jinsight**, and **Satin**, and the structure of the ownership tree in one of the simplest Java programs possible.

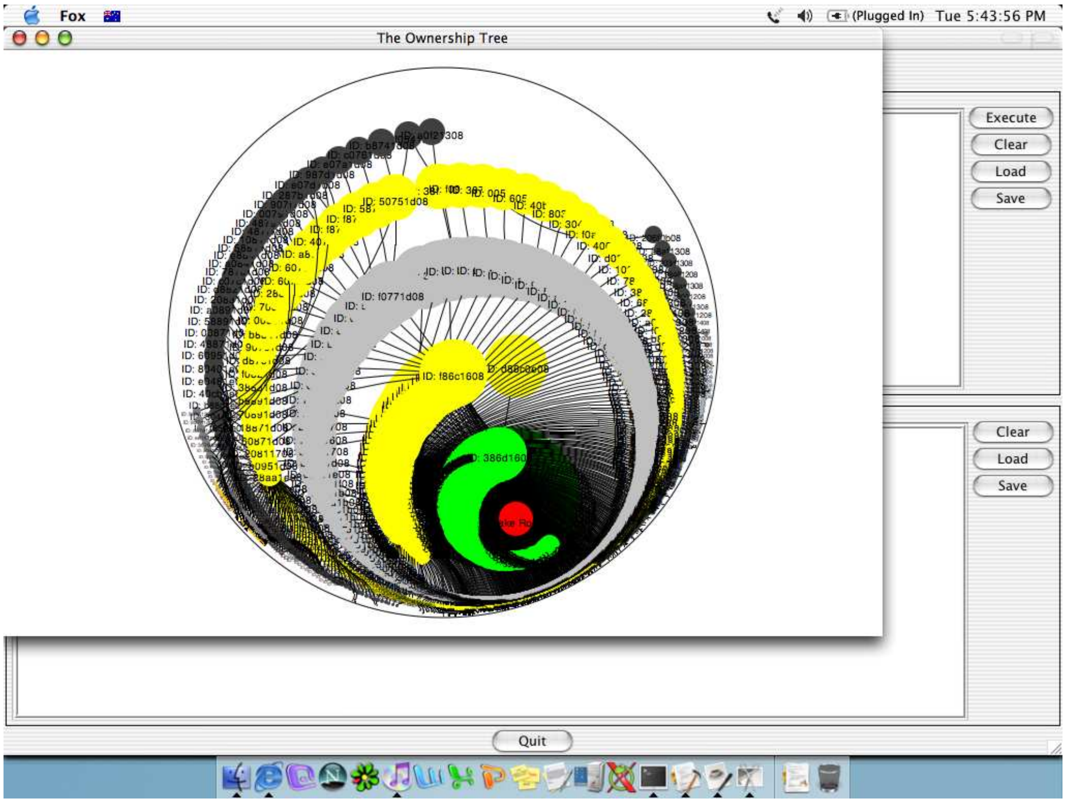


Figure 6.3: Visualising `HelloWorld` using Fish Eye View

6.2.1 Aliasing in ArgoUML

ArgoUML is a popular Java program for object-oriented modeling using UML. For example, figure 5.3, depicting the class structure underlying Fox, was created using ArgoUML. This program is open source and is available for free from `http://argouml.tigris.org/`. Figure 6.4 shows ArgoUML in action.

We ran ArgoUML and took snapshots of its memory at different stages during its run. The first one was taken while it was just starting to load. The second one was taking when it was about to finish loading. The third one was taken when it was running. The forth one was taken after using it in a normal way for a little while, working on a class diagram and a use case diagram. Finally, the fifth one was taken after we drove ArgoUML to the point when it was editing a large model and was using a large amount of memory.

After obtaining the five heap snapshots described above, we loaded each of them, one after another, using the queries of the following kind:

```
QLoadHeapSnapshot("~alex/Honours/Report/ArgoUML.StartingUp1.hprof");
```

For each of these snapshots, we wanted to find the percentage of objects having more than one incoming reference, which is conveniently described in FQL as follows:

```
QPercentage(``FIntegerProperty(
    ``PNumberOfDynamicIncomingReferences'',
    > 1, FSnapshot())'');
```

Which gives the percentage of aliased objects (i.e. those that have more than one incoming reference). Figure 6.5 shows the results obtained using this query.

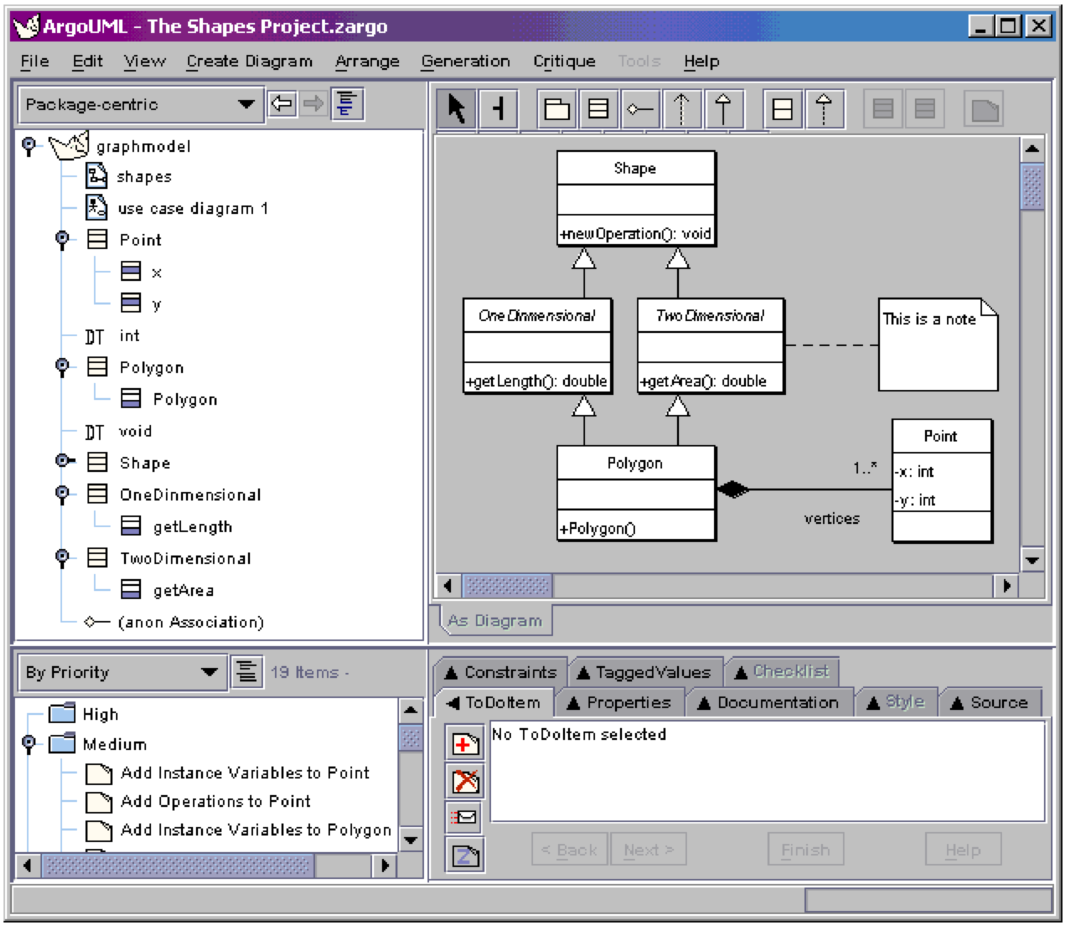


Figure 6.4: Screenshot of ArgoUML

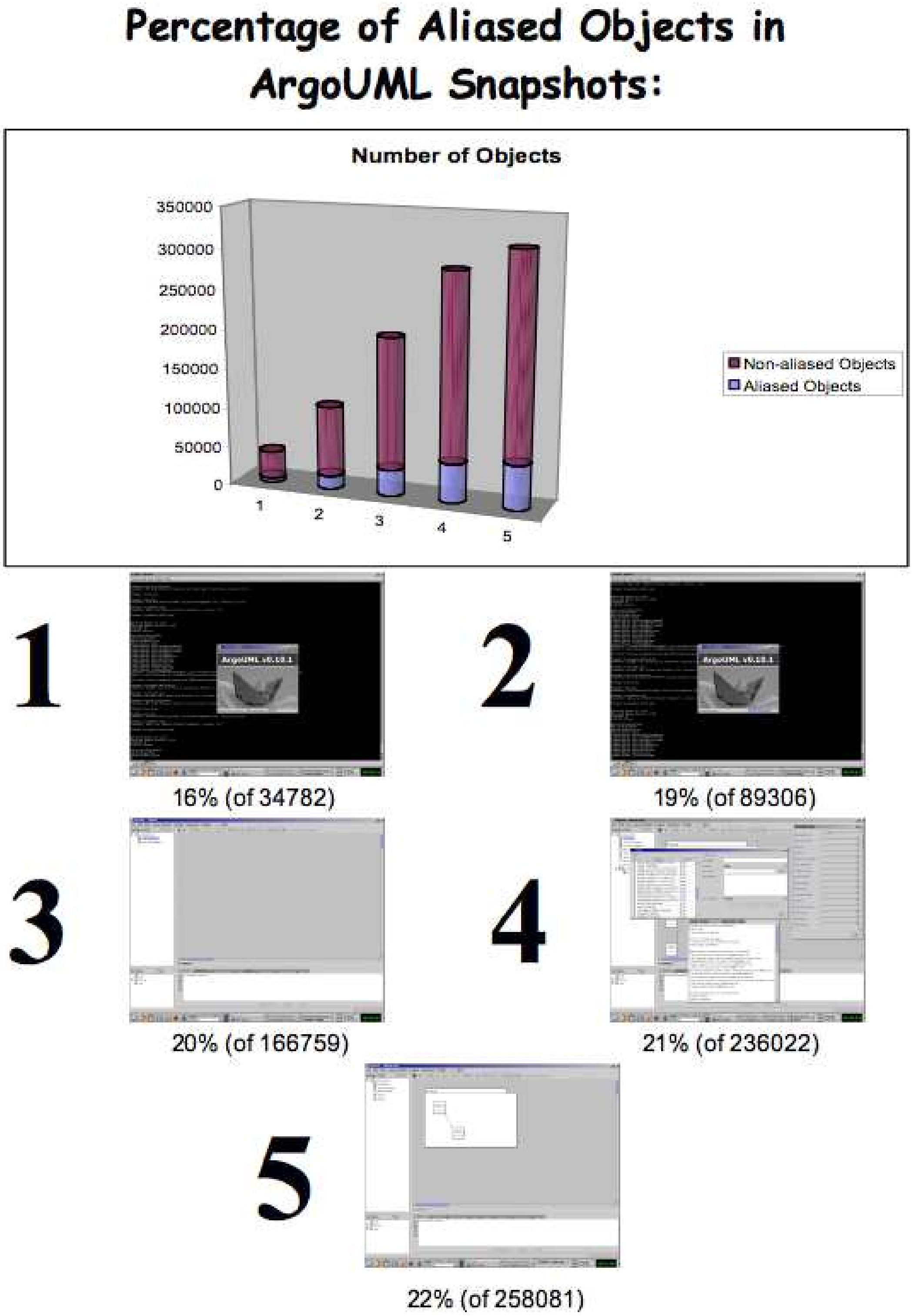


Figure 6.5: Aliasing at later and later stages of running ArgoUML

Initial snapshot taken as Fox just started.	23,736
Before loading a file with snapshot but after running for a little while.	23,809
After loading but still processing the snapshot.	231,618
Before destroying excess objects that are no longer required by setting their pointers to **null** and letting the garbage collector reclaim their memory.	723,788
After destroying and running for a little while.	99,733

Table 6.1: Five Snapshots of Fox Taken as it Processes Some Snapshot

We can see from the results that the percentage rose from 16%, up to 22% while at the same time the total number of objects rose from 34,782 to 258,081. This means that the number of aliased objects rose from 5,566 to 56,778. Every aliased object has a potential of introducing an error into the program, and a large number of them should indicate that the program's behaviour can be hard to predict.

6.2.2 The Fox Destroying its Internal Data Structures

We obtained five dumps of Fox itself as it was working with the heap snapshot of the simple `HelloWorld` program described above. Table 6.1 shows number of objects in each snapshot. We can see how the extra data structures used during the construction of the ownership tree vastly increase the size of Fox, while releasing memory used by no longer required data structures deallocates most of the objects letting the number of them drop from over 700,000 to less than 100,000.

6.3 Power Laws in Object Graphs

Lets consider a large program like **Forte**, which is Sun's Java Integrated Development Environment (IDE) written in Java. We took a snapshot of it running in a normal way. The snapshot had 359,666 objects as was discovered using the following query:

```
QCount("FSnapshot()");
```

Now, we consider the incoming references to each object. We use the following query to obtain an average number of incoming references per object in the object graph:

```
QAverage("FSnapshot()", "PNumberOfIncomingReferences");
```

This gives us:

```
Averaged 1.74 across the objects passing the filter.
```

Thus, a typical object will have around 1 or 2 incoming references. Now, what if we look at the number of objects that have just one incoming reference, the number of objects that have exactly two incoming references, and so on. We can obtain this information by saving the result of the following query into a comma-separated file and applying Microsoft Excel:

```
QProperties("PNumberOfIncomingReferences");
```

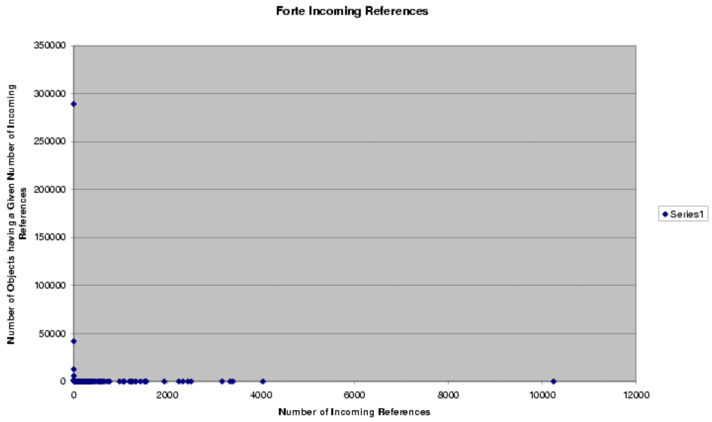


Figure 6.6: Incoming References in Forte

Figure 6.6 shows the graph of the number of objects having a particular number of incoming references versus the number of incoming references. This distribution follows a *Power Law* as shown below.

The *Power Law*, or Zipf's Law, was first observed by George Kingsley Zipf, a Harvard linguistics professor, in his study of the frequency of the words in English text. He observed that in the distribution of words in any given novel: the nth most common word occurs about $\frac{1}{n}$ times as frequently as the most common word.

In particular, the first power law observation was that if we take any English novel, count the number of times each word is used in it, and then sort them in order by putting the most frequently occurring word first. The rth ranked word in the list will occur $\frac{d}{r}$ times, where d is independent of r, but may be different from novel to novel.

The second power law observation was about the population of cities in a large country, like the United States. When we sort the cities in descending order by their population, the rth largest city will have the population about $\frac{d}{r}$, where d is independent of r but may vary from country to country [1].

Finally, a modern day example is the distribution of links to the web sites in the World Wide Web. Recent measurements have shown that probability of the page having i links to it from other pages around the web is:

$$Pr(indegree = i) \approx \frac{c}{i^{\beta}} \tag{6.1}$$

To show that in Forte, the distribution of incoming references follows a power law, we need to sort the objects in the order of the frequency of a particular number of incoming references occurring. Then we need to plot on the log-log scale the frequency rank of the number of incoming references occurrence versus the number of objects having such a rank. If the resulting graph forms a line, it shows a linear dependency between the rank and the number of objects of a particular rank. Figure 6.7 shows such a graph for five programs [2],

[1] This law usually doesn't hold for countries with centralised city planning.

[2] Please observe that although the data is discrete we have plotted it as a continuous line. This follows the tradition of power law papers.

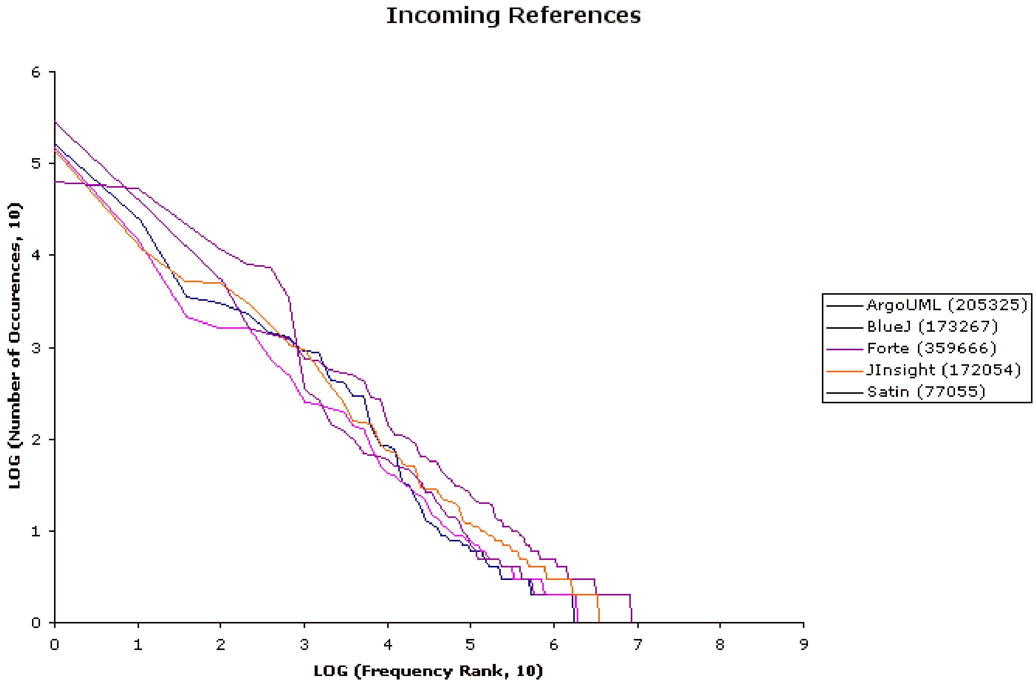


Figure 6.7: Distribution of incoming references across the five snapshots

including Forte. We can clearly observe that even though they all have a different number of objects inside the memory graph, the plots all lie on a line. Hence, we can conclude that the distribution of incoming references in the object graphs conforms to the Power Law.

Further examination of static incoming references and outgoing reference, or sizes of objects, in the object graph demonstrates two more Power Laws, as shown in figures 6.8 and 6.9.

The presence of a Power Law among the distributions of incoming and outgoing references shows that the vast majority of objects have a very small number of either incoming or outgoing references (exactly one, to be precise) and negligibly small number has more than three or four either incoming or outgoing references. Thus, most objects are unique (one incoming reference) and have size of around four bytes (one outgoing reference, which is a 32-bit address).

Finally, it would be interesting to look at whether there are any objects that have both a large number of incoming and outgoing references. Again, with the help of Microsoft Excel and Matlab, we were able to plot heat map representing the distribution of the number of objects having a particular number of incoming and outgoing references. Figure 6.10 shows such a heat map with darker colours corresponding to the smaller number of objects. We can see that the graph is very close to the two axis and no objects have a large number of both incoming and outgoing references. If we plot the two axis on a log-log scale, as shown in figure 6.11, we can see more closely that the majority of objects tend to have a small number of incoming references and a larger number of outgoing references and vice-versa [3].

[3]Note that the gap along the axis corresponds to the places where the LOG function is due to the approximation errors in Matlab of values close to zero

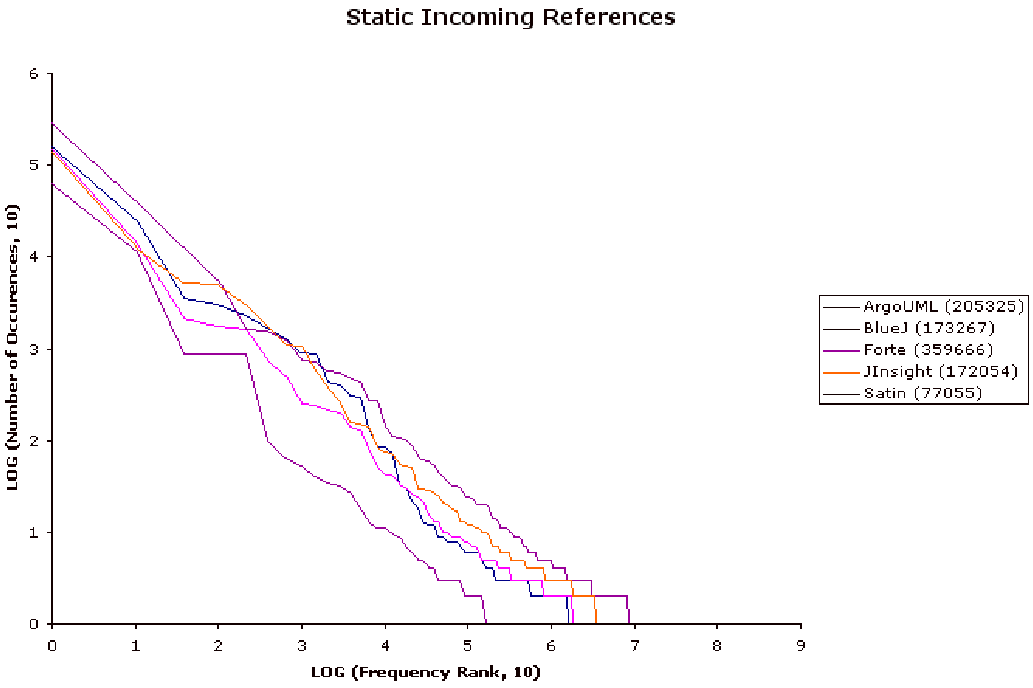


Figure 6.8: Distribution of static incoming references across the five snapshots

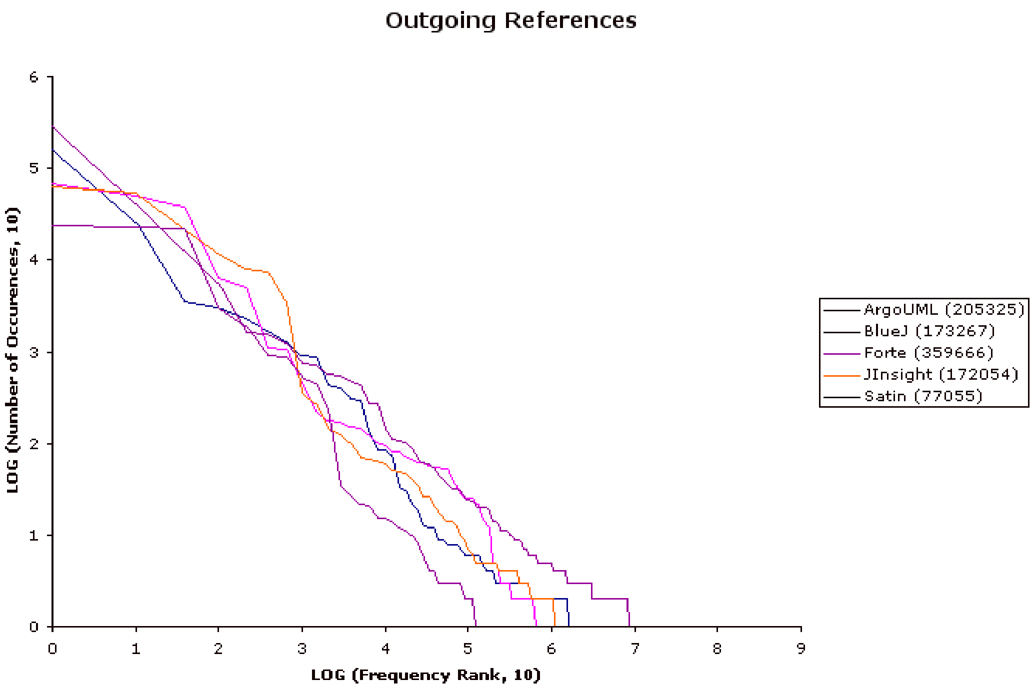


Figure 6.9: Distribution of outgoing references across the five snapshots

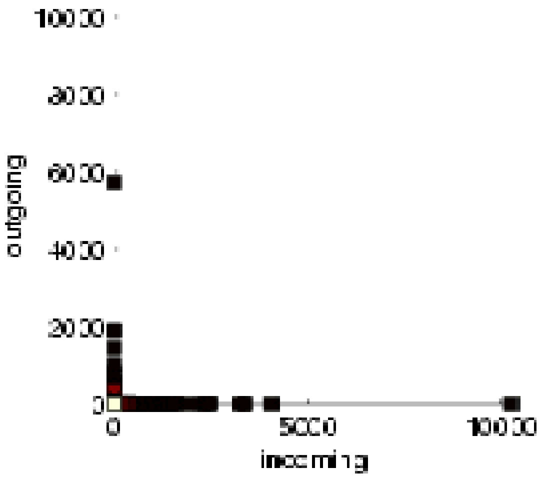


Figure 6.10: Incoming versus outgoing references in Forte

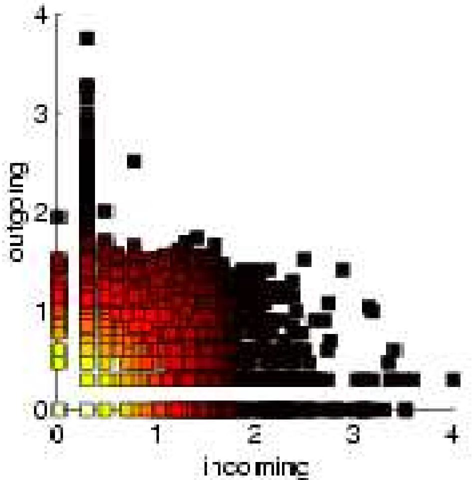


Figure 6.11: LOG of incoming versus LOG of outgoing references in Forte

6.4 Corpus Analysis

In this section, we describe the results of applying Fox to a large number of heap snapshots to examine a particular aspect of an object graph across a large number of programs. We have accumulated a corpus of 33 programs, half of which came from the Purdue Benchmark Suite used by Grothoff, Palsberg, and Vitek [7]. From these programs, we have obtained 52 heap snapshots across which we performed a number of measurements.

For each heap snapshot, we can calculate metrics such as the number of objects in the object graph, the number of unique roots in the Java heap root set, and the number of objects accessible by reference traversal from the root set (other objects are presumed to be uncollected garbage). These numbers give an idea of the size of the program under consideration. We found that a typical Java program in the corpus had around 60,000 objects allocated on the heap, about 1,800 of these were part of the Garbage Collector's root set (this includes static and global variables, metaobjects, and references from the Java stack) and 3,000 (around 5% or less) were garbage, i.e. not accessible from the root set.

In the sections below we present corpus-wide results on uniqueness, ownership, and confinement, followed by two comprehensive tables summarising the experiments that we performed upon the corpus.

6.4.1 Uniqueness

Uniqueness is the most basic type of aliasing control: a unique object is encapsulated within its sole referring object [3]. We analysed the object graphs to determine the number of non-unique objects, in particular, the percentage of objects with more than one incoming reference. We found that on average only 13% of all objects had more than one object pointing at them, or in other words had an alias. This means that the number of aliases in the systems we looked at was not too high. But further investigation showed that the percentage of aliased objects tends to increase as the program runs, stabilising at around 20%.

Given that the majority of objects are not aliased, we consider that it makes sense for formal techniques to support uniqueness. There are, however, enough aliased objects that uniqueness alone will be insufficient for managing aliasing within an object-oriented programming style.

6.4.2 Object Ownership

Object ownership is in essence a generalisation of uniqueness [20] that underlies many more sophisticated alias management schemes [17, 5, 4, 16]. An object a owns another object b if all the paths from the root r to the object b go through a. In this case b is called the *owner* of a. The implication of ownership is that no object outside the owner b is allowed to have a reference to a. We posit a global root r through which all the objects in the root set of the current heap snapshot can be accessed. Ownership allows us to structure an object graph into an implicit ownership tree.

Our primary metric of object ownership is the average depth of an object in the ownership tree, that is, the average numbers of levels of encapsulation around any object. In most programs in the corpus, this was around 5 or 6; however some large programs had substantially larger values (e.g. 817.25 under heavy load).

Given these figures, we hypothesised that large data structures such as linked lists could have a very significant effect when calculating the average depth of the node in the ownership tree: the average depth of a list node would be half the length of the list. To address this issue, we decided to fold up the reference paths by counting chains of objects of same class (e.g.

`LinkedList$Node`) as having the length of 1. This gave us a less biased account, with the average depth after folding being around 5.47 as opposed to 42.77 across all programs, with the large outlying depths being greatly reduced (e.g. a simple linked list test program has average depth of 142.37, average depth after folding of 4.21). The resulting ownership metric does demonstrate, however, that there is a significant amount of object-based encapsulation in Java programs.

We also calculated the average ownership tree depth per class; this gave us a finer-grained picture of object ownership. As an example of the average depth from the root per class, consider `java.lang.Hashtable`. Since most `java.util.Hashtable` instances should own (approximately) one `java.util.Hashtable$Entry[]` array instance, we would expect the average depth of the latter to be one greater than the former. The following sample of Rabbit output for **BlueJ** confirms this hypothesis, and also implies the actual hashtable entries are contained within the entry array.

```
Class java.util.Hashtable has 279 instances and on
average each instance node is 3.96 deep.
Class java.util.Hashtable$Entry[]; has 297 instances
and on average each instance node is 4.90 deep.
Class java.util.Hashtable$Entry has 3990 instances
and on average each instance node is 5.37 deep.
```

6.4.3 Confinement

The third aspect of encapsulation we analyse is object confinement, which is an instantaneous approach to class confinement [1, 7]. If all the referrers to an object are in the same package as the object's class, we call the object strongly confined. If all the referrers are in the same top level package, we call the object weakly confined: this is similar to the idea of hierarchical packages in [1, 7]. If there are referrers from a different package, we call it not confined. For example, if `java.util.Vector` object is pointed at by a `hat.model.Snapshot` object, then it is not confined. If all the referrers of `java.util.Vector` are members of `java.*` but not necessarily of `java.util.*` then it is weakly confined.

We have calculated a number of confinement metrics over the object graphs of the programs in our corpus, by considering all objects that refer to each object. We take an abductive view of class confinement: if the class is deduced to be confined to its package at all times during the program run [7], then all the instances of it should be confined.

Our measurements have shown that around 46% of objects were not confined, 21% were weakly confined, and 33% were strongly confined. At first glance, these numbers are roughly what static analysis by Grothoff, Palsberg, and Vitek [7] leads us to expect, even though we are looking from a different perspective.

We further analysed confinement on a per-class basis, rather than per-object. The second table in the end of this chapter gives the class confinement distribution. The first column, called 0%, lists the number of classes that have no instances that are strongly confined, and further columns list the number of classes that have a number of instances being strongly confined falling into a corresponding decile. The column named 100% counts those that have all their instances strongly confined.

One interesting feature of this table is that the 0% and 100% columns have by far the largest values: all the instances of a class tend to have the same confinement. We had hypothesised that there may be a significant fraction of the instances of some classes which were "almost" confined (90%), but this did not appear to be the case.

We can see that the vast majority of classes with instances present in our heap snapshot are not confined (88.48%). After comparing our results with the data obtained using the static control flow analysis by Grothoff, Palsberg, and Vitek [7] we noticed that the majority of classes that their tool detected as being confined at all times have not been instantiated in any of our heap snapshots. Those that were instantiated, we confirmed were confined from our perspective too.

6.4.4 General Collection of Metrics Across 52 Snapshots

Program	NO	NUR	ART	PMOIR	AD	ADF	NRK	NC	WC	SC	TT
Aglets	23212	1588	22723	10.64%	7.44	6.63	2502	37.62%	29.95%	32.43%	20 sec.
AlgebraDB	3749	481	3506	10.58%	6.08	4.32	568	36.85%	26.64%	36.51%	4 sec.
ArgoUML Heavy	128332	3710	123061	20.45%	20.20	4.95	12119	52.06%	20.77%	27.17%	302 sec.
ArgoUML Initial	99969	3553	95174	19.01%	9.25	4.93	9609	52.32%	21.10%	26.58%	212 sec.
ArgoUML Normal	207801	4129	201584	19.81%	19.05	5.13	16426	53.67%	21.17%	25.16%	560 sec.
ArgoUML	211625	4262	205325	19.50%	12.84	5.08	17172	53.66%	20.95%	25.39%	591 sec.
Bloat	3951	373	3834	4.72%	5.01	4.86	408	41.26%	23.32%	35.42%	4 sec.
BlueJ Heavy	175674	3821	173267	12.16%	817.25	6.71	12201	44.33%	23.32%	32.34%	994 sec.
BlueJ Initial	35955	2435	34639	12.73%	6.30	5.59	3688	41.52%	28.53%	29.94%	39 sec.
BlueJ Normal	101848	3194	99693	13.42%	718.47	6.22	8718	44.26%	24.62%	31.12%	530 sec.
BlueJ	34944	2455	33630	12.37%	6.16	5.66	3445	41.17%	28.85%	29.98%	36 sec.
DINO Heavy	30419	2364	29378	13.08%	6.53	5.60	3602	39.45%	28.23%	32.31%	30 sec.
DINO Initial	28813	2252	27824	12.68%	11.42	5.68	3245	38.42%	29.26%	32.32%	29 sec.
DINO Normal	29718	2287	28693	12.79%	6.81	5.65	3425	38.66%	28.34%	33.00%	29 sec.
DINO	29235	2205	28246	13.16%	9.26	5.67	3210	37.83%	28.88%	33.29%	28 sec.
DYNO	29183	2554	27806	16.41%	43.11	4.75	3824	41.28%	25.75%	32.97%	40 sec.
Denim	74044	3641	69550	14.57%	9.88	6.08	6312	39.16%	27.20%	33.64%	135 sec.
Doubly Linked List	3744	310	3627	31.76%	4.65	4.21	331	25.06%	23.57%	51.36%	4 sec.
Forte	374450	7503	359666	19.17%	40.77	6.15	37523	46.85%	20.79%	32.36%	2154 sec.
GJ	3146	391	2961	8.14%	5.08	4.53	422	34.38%	30.87%	34.75%	3 sec.
HAT	258959	486	256290	15.64%	5.83	5.74	1205	26.88%	13.44%	59.67%	679 sec.
HelloWorld	2746	309	2633	5.73%	4.90	4.68	330	34.68%	32.59%	32.74%	3 sec.
HyperJ	3327	498	3194	6.64%	4.65	4.46	538	37.54%	31.65%	30.81%	3 sec.
JAX	33467	2600	32095	17.32%	5.67	4.83	5753	43.26%	21.60%	35.14%	39 sec.
JDI Heavy	34166	2404	33024	13.72%	12.44	5.76	3810	42.38%	27.02%	30.60%	36 sec.
JDI Initial	33123	2391	32119	13.45%	16.03	5.59	3734	39.95%	28.19%	31.86%	33 sec.
JDI Normal	52540	2359	51396	14.51%	9.00	6.06	4556	51.40%	21.42%	27.18%	54 sec.
JDI	35114	2431	34098	14.82%	59.21	5.51	3747	38.70%	28.85%	32.45%	43 sec.
Jinsight Heavy	172987	2437	172054	18.19%	18.10	4.31	29253	68.09%	18.94%	12.98%	356 sec.
Jinsight Initial	24190	1912	23317	10.15%	6.54	6.11	2410	38.41%	30.90%	30.69%	23 sec.
Jinsight Normal	101565	2363	100616	8.67%	9.05	5.01	7256	67.04%	16.08%	16.88%	123 sec.
Jinsight	63053	2267	62136	18.34%	15.92	4.82	10376	59.42%	20.81%	19.77%	80 sec.
JTB	3261	434	3128	5.66%	4.63	4.44	464	36.41%	28.55%	35.04%	3 sec.
Jess	10685	580	10487	8.71%	6.13	6.01	1082	45.04%	13.72%	41.24%	8 sec.
Jython	27458	1406	26739	11.69%	8.51	8.26	2573	35.35%	16.23%	48.41%	24 sec.
Kacheck/J Heavy	120286	425	119979	14.09%	10.54	5.53	10837	31.52%	10.01%	58.47%	121 sec.
Kacheck/J Initial	8043	392	7910	3.01%	8.50	7.76	437	27.02%	42.97%	30.01%	7 sec.
Kacheck/J Normal	24345	426	24058	10.97%	6.73	6.05	2084	29.50%	20.12%	50.38%	19 sec.
Kacheck/J	9372	413	9169	5.98%	7.88	7.14	743	28.90%	38.08%	33.01%	8 sec.
Kawa	9685	805	9423	11.80%	4.81	4.14	1148	48.64%	10.84%	40.53%	8 sec.
Linked List	3744	310	3627	4.16%	142.37	4.21	331	25.06%	23.57%	51.36%	6 sec.
Log File System	27111	2123	26224	11.84%	23.85	5.83	2900	39.64%	29.47%	30.90%	43 sec.
OVM	9372	413	9169	5.98%	7.88	7.14	743	28.90%	38.08%	33.01%	8 sec.
Ozone	16031	660	15474	11.65%	7.05	6.42	2037	33.48%	31.09%	35.43%	14 sec.
SableCC	21486	515	20989	16.68%	5.73	4.67	1561	57.88%	11.97%	30.15%	16 sec.
Satin	80415	1922	77055	18.52%	8.83	4.40	13152	48.47%	23.55%	27.99%	174 sec.
Schroeder	101659	1432	36371	17.55%	9.04	5.29	3221	41.75%	27.28%	30.98%	891 sec.
Skyline	2106	192	2071	5.12%	4.87	4.62	206	34.09%	33.61%	32.30%	2 sec.
Soot	11588	583	11309	16.82%	4.90	4.75	1931	46.53%	17.28%	36.19%	10 sec.
SwingSet	49343	2889	47855	16.64%	7.35	5.48	5327	46.14%	20.63%	33.23%	57 sec.
Toba	2920	328	2828	6.30%	5.13	4.83	377	32.32%	34.19%	33.49%	3 sec.
Tomcat	34688	1025	33873	9.38%	6.41	6.07	2824	39.08%	26.10%	34.81%	32 sec.
Across All	58050.9	1793.04	55286.42	12.82%	42.77	5.47	5301.85	46.03%	21.29%	32.68%	167 sec.

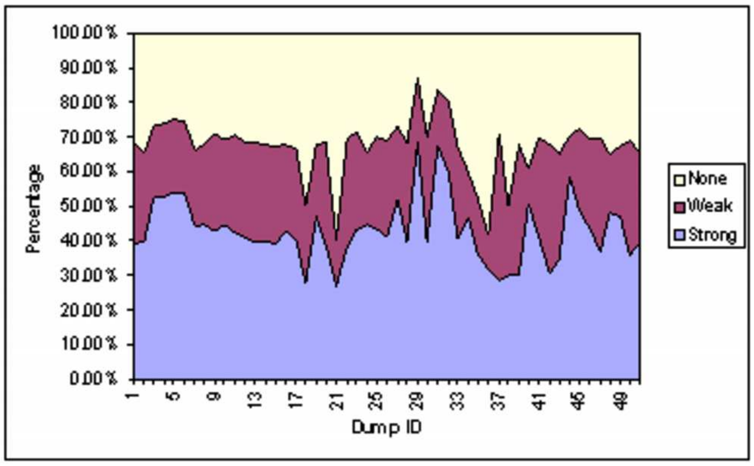
1. **NO** - Number of Objects in the Object Graph
2. **NUR** - Number of Unique Roots in the Heap Root Set
3. **ART** - Number of Objects Accessible by Reference Traversal from the Root Set
4. **PMOIR** - Percentage with MORE THAN ONE Incoming Reference
5. **AD** - Average Depth from the Root in the Tree
6. **ADF** - Average Depth from the Root in the Tree After Folding
7. **NRK** - Number of Root Kids in the Resulting Ownership Tree
8. **NC** - Percentage not Confined
9. **WC** - Percentage Weakly (but not Strongly) Confined
10. **SC** - Percentage Strongly Confined
11. **TT** - Time Taken to Analyse a Given Heap Snapshot [4]

[4] The Machine used was a Sun Ultra80-450 with 4x450MHz UltraSPARC II, 4MB Cache and 2048MB RAM.

6.4.5 Class Confinement Metrics Across 52 Snapshots

Program	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	99%	100%	TNC
Aglets	5729	7	6	3	5	7	0	2	0	6	4	757	6526
AlgebraDB	1007	1	1	0	1	1	0	2	0	1	1	272	1287
ArgoUML Heavy	9171	24	3	8	15	15	3	10	4	7	8	1620	10888
ArgoUML Initial	8763	22	5	6	13	14	3	10	4	7	5	1551	10403
ArgoUML Normal	9895	25	3	10	14	14	2	10	3	7	9	1629	11621
ArgoUML	9975	23	8	7	14	14	2	11	4	7	8	1677	11750
Bloat	1313	4	1	1	2	3	0	0	0	2	0	244	1570
BlueJ Heavy	18908	28	10	8	17	17	3	10	8	9	14	1847	20879
BlueJ Initial	7528	18	4	7	12	16	4	5	6	2	6	1264	8872
BlueJ Normal	17382	25	6	7	12	17	3	13	9	8	12	1595	19089
BlueJ	7364	17	4	6	12	13	4	5	6	2	6	1263	8702
DINO Heavy	5876	16	3	5	11	12	3	7	4	4	7	1306	7254
DINO Initial	5709	14	3	3	10	15	4	5	4	3	7	1271	7048
DINO Normal	5817	14	3	4	11	13	3	5	5	3	7	1291	7176
DINO	5732	14	3	3	9	18	2	6	5	3	7	1251	7053
DYNO	5277	15	7	5	11	14	2	7	5	1	8	1429	6781
Denim	10529	21	10	5	10	14	3	11	6	6	9	1731	12355
Doubly Linked List	863	2	0	0	1	3	0	0	0	1	1	213	1084
Forte	48546	53	18	12	27	38	10	11	7	15	18	1831	50586
GJ	939	2	1	3	0	1	0	1	0	1	0	232	1180
HAT	10298	5	1	1	2	3	1	1	0	0	1	324	10637
HelloWorld	864	2	0	0	2	2	0	0	0	1	0	212	1083
HyperJ	1155	1	3	0	0	3	0	0	1	2	0	255	1420
JAX	6813	13	10	5	9	12	4	6	10	3	3	1415	8303
JDI Heavy	6830	17	6	5	14	13	3	6	5	3	6	1333	8241
JDI Initial	5906	18	4	4	13	10	2	9	2	2	8	1345	7323
JDI Normal	9283	18	6	5	14	16	2	6	5	3	6	1322	10686
JDI	5918	18	7	4	9	9	2	8	4	2	7	1371	7359
Jinsight Heavy	11707	16	10	3	9	11	3	3	3	5	5	1081	12856
Jinsight Initial	5566	10	6	2	9	11	1	3	2	2	6	997	6615
Jinsight Normal	10710	17	6	5	11	12	3	2	2	3	4	1118	11893
Jinsight	7676	16	9	3	9	13	2	5	2	4	5	1078	8822
JTB	1051	1	2	0	0	2	0	0	0	2	0	255	1313
Jess	1815	5	0	2	0	2	0	2	4	0	4	350	2184
Jython	4128	5	1	0	0	5	1	1	2	1	8	292	4444
Kacheck/J Heavy	10180	6	1	3	1	0	0	0	0	0	6	467	10664
Kacheck/J Initial	1779	5	0	1	0	2	0	0	0	0	1	257	2045
Kacheck/J Normal	3414	6	1	3	0	1	0	0	0	0	6	468	3899
Kacheck/J	2281	6	1	0	2	1	0	0	0	0	1	432	2724
Kawa	2243	3	2	4	2	2	0	2	4	2	5	301	2570
Linked List	863	2	0	0	1	3	0	0	0	1	1	213	1084
Log File System	5517	12	5	4	11	11	1	5	3	4	8	1168	6749
OVM	2281	6	1	0	2	1	0	0	0	0	1	432	2724
Ozone	3787	6	2	1	4	6	0	3	1	1	2	395	4208
SableCC	2916	8	1	1	1	3	2	2	0	2	4	439	3379
Satin	7088	10	9	4	1	5	0	4	4	1	9	854	7989
Schroeder	6912	11	10	4	16	11	4	8	4	9	8	255	7252
Skyline	700	1	1	0	0	1	0	0	0	0	0	122	825
Soot	3471	5	3	2	1	5	0	4	0	1	2	317	3811
SwingSet	7179	16	10	10	14	13	2	6	1	3	7	1484	8745
Toba	833	4	1	3	1	1	0	1	0	0	0	213	1057
Tomcat	7771	10	8	3	3	11	0	6	5	1	4	482	8304
Total	345258	624	226	185	368	460	84	224	144	153	265	45321	393312
Percentage	87.78%	0.16%	0.06%	0.05%	0.09%	0.12%	0.02%	0.06%	0.04%	0.04%	0.07%	11.52%	100.00%

Strong/Weak/Not Confined Distribution Across the Snapshots



Chapter 7

Conclusion

In this report we have presented Fox, a tool to study aliasing in object graphs of Java programs. We described the query language, called FQL, that lies at its foundation. Finally we presented the results of our studies of object graphs that utilised Fox and FQL. The tool proved very useful and we plan to continue its application in the future work, as described below.

7.1 Contributions

We provided the aliasing research community with a new tool that can be used for static heap analysis. We developed a useful query language that can be extended as further studies are performed by us and other people. Finally, we performed a large number of experiments as presented in this report that directly involved using Fox.

The results described in this report included the demonstration of the Power Law dependency among the references in object graphs, empirical confirmation of the large amount of aliasing in Java programs, and a closer examination of ownership and confinement among a large number of Java programs. In particular, we found that on average objects were five layers deep within an object ownership hierarchy and that of all objects one third were strongly confined according to our instantaneous confinement definition. One of the contributions to the metrics themselves was our development and use of the concept of *folding* in the ownership tree which demonstrated a more sensible measure of object's encapsulation.

Fox and our corpus of heap snapshots are available at: `www.mcs.vuw.ac.nz/~alex/fox`.

7.2 Future Work

In the future, we plan to concentrate on exploration of different interesting properties of ownership, trying to answer questions like: "are there objects that only have one pointer from outside their ownership shadow?", or "are there objects that have a large number of incoming references, and only one child that has one incoming reference but a large number of outgoing references?". We plan to examine the formal research done in the area of aliasing and try and apply our tool to see the extent of applicability of formal results.

The work described above, should help us improve and develop further the query language. We have two different ways to go from where we are now: we can either extend the FQL and make its syntax nicer and more compatible with Object Constraint Language (OCL), or we can decide to avoid doing storage of heap snapshots ourselves and use Fox to convert memory snapshots into appropriate database, which we can then query extensively using standard Structured Query Language (SQL).

One of the future obvious extensions to Fox itself would be duplication of some of the control-related queries in the user interface to simplify the task of a user using Fox for a detailed analysis of a single heap snapshot only.

Once the tool matures, we plan to make it available on open source basis to all the researchers worldwide in the hope that it will advance the study of aliasing using static analysis of complete heap snapshots.

Bibliography

[1] Boris Bokowski and Jan Vitek. Confined types. In *Proceedings of OOPSLA'99*, ACM Press, 1999.

[2] Grady Booch. Object-oriented design with applications. The Benjamin/Cummings Publishing Company, 1991.

[3] John Boyland, James Noble, and William Retert. Capabilities for sharing. In *Proceedings of ECOOP'01*, Springer-Verlag, 2001.

[4] David Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *Proceedings of ECOOP'01*, Springer-Verlag, 2001.

[5] David Clarke, John Potter, and James Noble. Ownership types for flexible aliasing protection. In *Proceedings of OOPSLA'98*, ACM Press, 1998.

[6] Bill Foote. Java Heap Analysis Tool. Available at: *http://java.sun.com/people/billf/heap/index.html*

[7] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of OOPSLA'01*, ACM Press, 2001

[8] D. E. Harms and B. Weide. Copying and swapping: influences on the design of reusable software components. In *IEEE Transactions of Software Engineering*, 17(5): 424-435.

[9] Trent Hill, James Noble, John Potter. Scalable visualisations with ownership trees. In *Proceedings of TOOLS Pacific 2000, Sydney, Australia*, IEEE CS Press, 2000.

[10] C.A.R. Hoare and He Jifeng. A trace model for pointers and objects. In *Proceedings of ECOOP'99*, 1999.

[11] Chanika Hobatr and Brian A. Malloy. The design of an OCL query-based debugger for C++. In *Proceedings of 16th ACM SAC2001 Symposium on Applied Computing*, 2001.

[12] IBM AlphaWorks. Jinsight. Available at: *http://www.alphaworks.ibm.com/tech/jinsight/*

[13] James L. Johnson. *Database: models, languages, design.* Oxford University Press, 1997.

[14] Raimondas Lencevicius. *Advanced debugging methods.* Kluwer Academic Publishers, August 2000.

[15] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. In *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[16] P. Muller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffer and J. Meyer editors, *Programming Languages and Fundamentals of Programming*, Fernuniversitat Hagen, 1999.

[17] James Noble, John Potter, Jan Vitek. Flexible alias protection. In *Proceedings of ECOOP'98*, 1998.

[18] Open Virtual Machine for Java. Available at: *http://www.ovmj.org/*

[19] Alex Potanin and James Noble. Checking ownership and confinement properties. In *Formal Techniques for Java-like Programs Workshop at ECOOP'02*, 2002.

[20] John Potter, James Noble, and David Clarke. The ins and outs of objects. In *Proceedings of Australian Software Engineering Conference (ASWEC)*, IEEE CS Press, 1998.

[21] Ehud Y. Shapiro. *Algorithmic program debugging.* The MIT Press, 1983.

[22] G. Santucci, P. A. Sottile. Query by diagram: a visual environment for querying databases. In *Software Practice and Experience*, Vol. 23, No. 3, 1993.

[23] Secure Internet Programming Group. Available at: *http://www.cs.princeton.edu/sip/news/april29.html*

[24] Sun Microsystems. Java Virtual Machine Profiler Interface. Available at: *http://java.sun.com/j2se/1.4/docs/guide/jvmpi/index.html*

[25] VisualBeans.Com. FishEye Bean. Available at: *http://www.visualbeans.com/FishEye/*

[26] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: a responsibility-driven approach. In *Proceedings of OOPSLA'89*, 1989.

[27] Andreas Zeller. Delta debugging. Available at: *http://www.st.cs.uni-sb.de/dd/*

[28] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, 2002.

[29] T. Zimmermann, A. Zeller. Visualizing memory graphs. In *Proceedings of the Dagstuhl Seminar 01211 "Software Visualization"*, Lecture Notes in Computer Science, Dagstuhl, Germany, Springer-Verlag, May 2001.

[30] M. Zloof. Query-by-example: a database language. In *IBM Systems Journal*, 16(4):324–343, 1977.