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 Object Graph Analysis

Alex Potanin and James Noble and Robert Biddle
{alex, kjx, robert}@mcs.vuw.ac.nz

Technical Report CS-TR-03/15
December 2003

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 Object Graph Analysis

Alex Potanin and James Noble and Robert Biddle
{alex, kjx, robert}@mcs.vuw.ac.nz

Technical Report CS-TR-03/15
December 2003

Abstract

Object-oriented programs, when executed, produce a large number of objects in memory. These objects and the references between them form an *object graph.* Object graphs are not obvious from the class structure underlying them, and thus it is important to study them to better understand what really happens to programs at run time. We present a query-based debugger called Fox that allows a detailed examination of snapshots taken of object graphs as programs run. Fox makes it easy to examine object structures, encapsulation and ownership properties, and it has proven essential both in our research and day to day programming.

Publishing Information

Unpublished.

Author Information

Alex Potanin is a PhD Student supervised by James Noble and Robert Biddle

The Fox — A Tool for Object Graph Analysis

Alex Potanin **James Noble** **Robert Biddle**

School of Mathematical and Computing Sciences
Victoria University of Wellington, New Zealand
`{alex, kjx, robert}@mcs.vuw.ac.nz`

Abstract

Object-oriented programs, when executed, produce a large number of objects in memory. These objects and the references between them form an *object graph*. Object graphs are not obvious from the class structure underlying them, and thus it is important to study them to better understand what really happens to programs at run time. We present a query-based debugger called Fox that allows a detailed examination of snapshots taken of object graphs as programs run. Fox makes it easy to examine object structures, encapsulation and ownership properties, and it has proven essential both in our research and day to day programming.

Keywords: object graph analysis, query-based debugging

1 Introduction

As computers grew in computational power, so did the complexity of software that came with them. To help manage the growing size of software development projects, researchers developed the concept of *object-oriented programming*. It proved to be extremely popular and has been widely adopted by the programming community. Programs are now structured as collections of classes that model 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 they are free of errors and other defects.

In this paper, we present a tool called the Fox that we developed to be able to systematically query and examine in detail the object behaviour in real life programs. We used it to perform a large scale corpus analysis (Potanin, Noble & Biddle 2003) and to discover that object graphs are scale-free (Potanin, Noble, Frean & Biddle 2003). In this paper we describe how Fox was designed and how it can be used by developers and researchers to provide insight into the run time structure underlying modern software.

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 could 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.

In the next section, we give a brief introduction to the area of object graph analysis and present the related work. Section 3 describes our tool and a query language that it uses. Section 4 shows a small selection of examples that demonstrate how Fox can be used. Finally, in section 5 we discuss how the Fox proved to be essential for our research and how we hope that it continues to contribute as a publically available tool.

2 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 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 a `LinkedList` object 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. In running programs, object graphs have one or more *roots* (such as threads, or global or stack variables) representing objects that are permanently accessible by the current computation: objects that become unreachable from the roots are considered garbage and any memory they occupy may be reused.

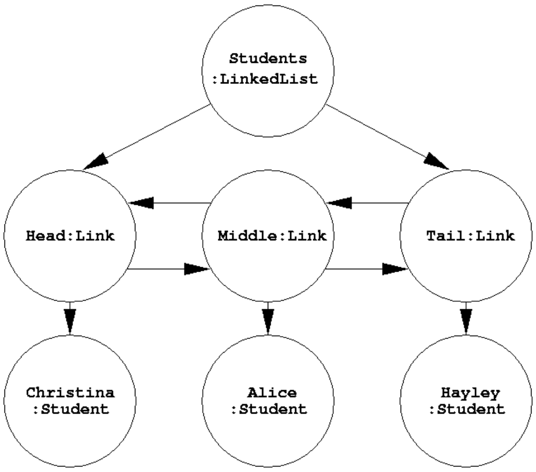


Figure 1: Simple object graph of a linked list

Object graphs are the most fundamental structure in object-orientation. The primary aim of object-oriented analysis is to model the real world in terms of communicating objects (that is, in terms of an object graph), while object-oriented design produces a description of an object graph that will eventually be embodied in a program. The artifacts and methods of object-orientation (classes, associations, interfaces, inheritance, packages, patterns, UML, CRC Cards, and so on) are ultimately techniques for defining object graphs by describing the contents of the objects and the structure of the links between them.

2.1 Analysing Object Graphs

The study of memory structures is a nontrivial task. One of the motivations is that it 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.

Analysis approaches include dynamic visualisation of the inter-object relationships as the program runs (Zimmermann & Zeller 2001, Hill, Noble & Potter 2000), code analysis predicting all possible object graphs resulting from the underlying class structure (Grothoff, Palsberg & Vitek 2001), and query-based debuggers that verify that certain inter-object relationships hold throughout the graph existence (Lencevicius 2000). 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.

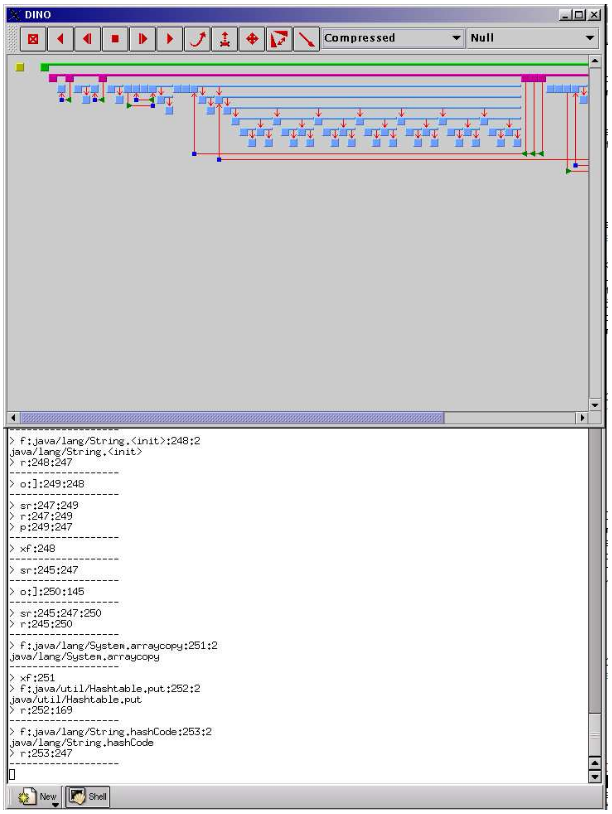
When we started our work on this project, our aim was to find a tool that would allow us to measure encapsulation aspects of real object graphs inside running programs. Unfortunately, none of the tools we examined could effectively estimate uniqueness, ownership or confinement parameters. For this reason, we decided to develop our own object graph analysis tool. We decided to analyse Java programs, because there are a large number of Java programs available on the Internet, because Java is straightforward to analyse compared with hybrid object-oriented languages such as C++, and because Sun's Java Developer Kit (JDK) (Sun Microsystems 2002*a*) includes a number of standardised interfaces and libraries to support tool development.

The rest of this section presents a small selection of tools we looked at that are designed to assist software engineering researchers and programmers in object graph analysis. We present a dynamic visualisation tool, a static analysis tool, and a debugging tool — all great examples of a vast number of object graph analysis software available. We then show a query-based debugger and a number of tools for detailed analysis of a static snapshot of a running program. An idea of combining the query language flexibility and the completeness of information contained in a heap dump, lies at the foundation of our tool.

2.2 DINO

DINO is an ownership tree visualisation tool written by Trent Hill (Hill et al. 2000). An object a owns another object b if all the paths from the root of the graph to b go through a. The concept of *ownership* allows us to structure an object graph into an ownership tree, where we can guarantee that only objects below, or at the same level as some object in the tree can have direct references to it (Noble, Vitek & Potter 1998). Figure 2 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.

One of the issues the developers of DINO had to face is inability to visualise every single object that exists during the execution of a program, because of the vast number of them. Hence, DINO allows the user to concentrate only on the user-created objects that are relevant to the code being executed, and avoid looking at the internal processes inside system libraries which may or may not have an adverse effect on the code in question.



```
--------------------
> f:java/lang/String.<init>:248:2
java/lang/String.<init>
> r:248:247
--------------------
> o:]:249:248
--------------------
> sr:247:249
> r:247:249
> p:249:247
--------------------
> xf:248
--------------------
> sr:245:247
--------------------
> o:]:250:145
--------------------
> sr:245:247:250
> r:245:250
--------------------
> f:java/lang/System.arraycopy:251:2
java/lang/System.arraycopy
--------------------
> xf:251
> f:java/util/Hashtable.put:252:2
java/util/Hashtable.put
> r:252:169
--------------------
> f:java/lang/String.hashCode:253:2
java/lang/String.hashCode
> r:253:247
--------------------
```

Figure 2: DINO — an ownership tree visualiser

2.3 Kacheck/J

Kacheck/J (Grothoff et al. 2001) 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.

Although the tool can properly detect classes which are guaranteed to have all instances properly confined throughout the execution, it is not designed to look at a program as it executes (and thus allow to find objects that are confined at a particular instance) and it doesn't deal with alternative approaches to restricting the pointers allowed between objects, such as ownership.

2.4 HOWCOME and Delta Debugging

HOWCOME is a cause-effect gap detector written by Zeller (Zeller 2002). It constitutes a part of the work on the Delta Debugging Project (Zeller 2003).

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.

2.5 Query-Based Debuggers

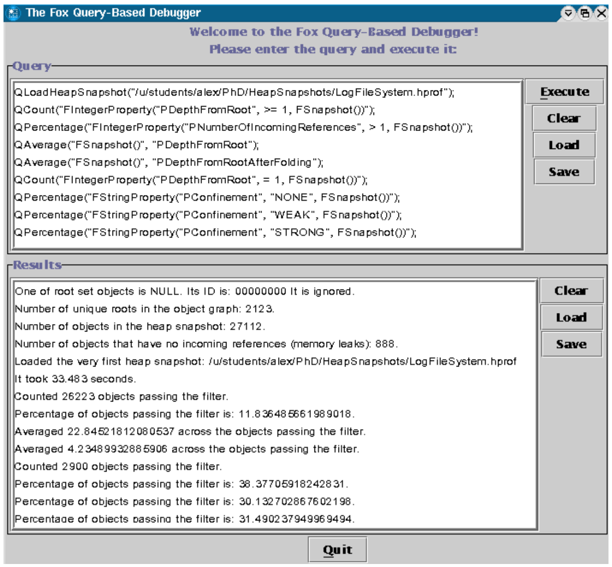
Lencevicius's work in query-based debugging (Lencevicius 2000) allows a programmer to either dynamically (as the program is running) or statically (as the program's memory snapshot is considered) verify relationships between objects. For example, the following query will verify if all the nodes in a linked list point to different elements:

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

This 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.

2.6 HAT: The Heap Analysis Tool

The Java Virtual Machine Profiler Interface (JVMPI) (Sun Microsystems 2002*b*) provides low-level hooks directly into the structures of the virtual machine. Part of this interface allows the user to save a snapshot of the entire object graph of any Java program into a file. These heap snapshots can be inspected using the Heap Analysis Tool (HAT) written by Bill Foote (Foote 2002). Using a web browser as a user interface, the HAT supports simple interaction with an object graph — navigating links forwards and backwards between objects, enumerating all the instances of a given class, and so on.



```
QLoadHeapSnapshot("/u/students/alex/PhD/HeapSnapshots/LogFileSystem.hprof");
QCount("FIntegerProperty("PDepthFromRoot", >= 1, FSnapshot())");
QPercentage("FIntegerProperty("PNumberOfIncomingReferences", > 1, FSnapshot())");
QAverage("FSnapshot()", "PDepthFromRoot");
QAverage("FSnapshot()", "PDepthFromRootAfterFolding");
QCount("FIntegerProperty("PDepthFromRoot", = 1, FSnapshot())");
QPercentage("FStringProperty("PConfinement", "NONE", FSnapshot())");
QPercentage("FStringProperty("PConfinement", "WEAK", FSnapshot())");
QPercentage("FStringProperty("PConfinement", "STRONG", FSnapshot())");
```

```
One of root set objects is NULL. Its ID is: 00000000 It is ignored.
Number of unique roots in the object graph: 2123.
Number of objects in the heap snapshot: 27112.
Number of objects that have no incoming references (memory leaks): 888.
Loaded the very first heap snapshot: /u/students/alex/PhD/HeapSnapshots/LogFileSystem.hprof
It took 33.483 seconds.
Counted 26223 objects passing the filter.
Percentage of objects passing the filter is: 11.836485661989018.
Averaged 22.845218120080537 across the objects passing the filter.
Averaged 4.234899328885906 across the objects passing the filter.
Counted 2900 objects passing the filter.
Percentage of objects passing the filter is: 38.3770591824283 1.
Percentage of objects passing the filter is: 30.132702867602198.
Percentage of objects passing the filter is: 31.490237949969494.
```

Figure 3: The Fox Query-Based Debugger

2.7 Jinsight

Jinsight (AlphaWorks 2003) is written by IBM Research 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.

Jinsight clearly shows an advantage of having a complete information about the current state of the program, to be able to perform a thorough analysis.

2.8 Leakbot

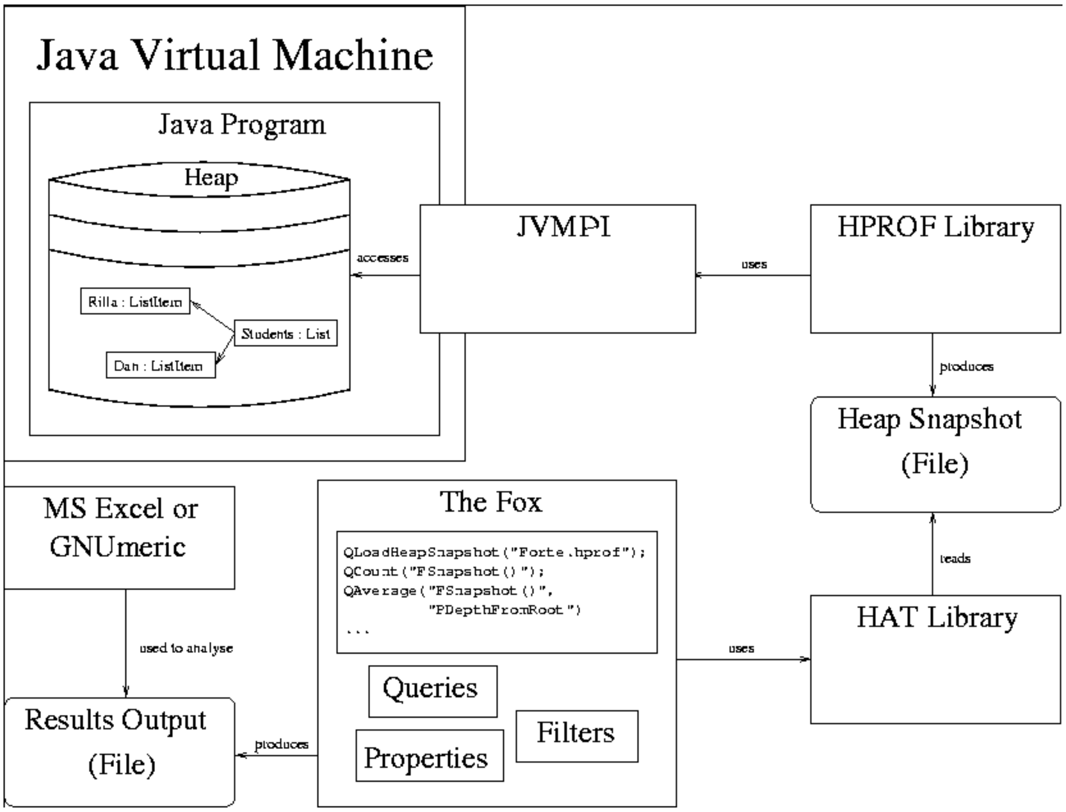
Leakbot (Mitchell & Sevitsky 2003) was developed as part of the Jinsight project at IBM Research and it provides a user with semi-automated way of detecting memory leaks. It takes an object graph snapshot of a running program and creates an ownership tree to be able to rate each object (based on its position inside the ownership tree and a number of other factors) with respect to its potential to become a memory leak.

Yet again, it shows that examination of the run time structure underlying object-oriented programs in great detail can provide valuable insights.

3 The Fox

In 2002, we wrote a tool (named Rabbit) that used information from HAT about the objects to construct an ownership tree from the object graph. Rabbit used a well known dominator tree construction algorithm (Lengauer & Tarjan 1979) to construct the ownership tree, and then calculated a range of uniqueness, ownership, and confinement metrics. Each calculation was specified in a separate class: a new calculation had to be implemented by providing a new `Calculation` subclass and recompiling Rabbit. The results Rabbit was allowing us to get proved so valuable that we decided to develop a more general query-based debugger, called Fox.

The need to recompile Rabbit, debug a new extension, and then reanalyse a corpus of object graph snapshots, meant that Rabbit was not particularly effective for exploratory analysis of object graphs. After examining the related research in object graph analysis using query-based debuggers, especially the work by Lencevicius (Lencevicius 2000), we decided to improve the flexibility of our tool by introducing a query language.



```
QLoadHeapSnapshot("Forte.hprof");
QCount("FSnapshot()");
QAverage("FSnapshot()",
         "PDepthFromRoot")
...
```

Figure 4: Architecture of Fox

Figure 3 shows the final tool that we developed, called Fox, as a successor to Rabbit. Fox's query language, called FQL (Fox Query Language), can express a range of calculations without rebuilding the program or reloading object graph snapshots.

Once the information about a heap snapshot is loaded into memory, Fox views it as a single database table with each row containing information about a single object. We calculate a number of *properties* for each object and store them as columns in the table. FQL *filter* statements select objects meeting a set of constraints on their *properties*, and *queries* report information to the user.

FQL queries include standard 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 that visualise part of the current object graph.

For example, to determine the percentage of objects having more than one incoming reference in a snapshot, we would run the following FQL query:

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

3.1 Architecture

Figure 4 depicts the architecture of our approach to the study of memory structures. Consider a Java program being executed inside a Java Virtual Machine (JVM). We use the Heap Profiler (HPROF) Library that utilises the Java Virtual Machine Profiler Interface (JVMPI) (Sun Microsystems 2002*b*) 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 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 performed with Fox 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.

3.2 The Fox Query Language

The study of object graphs is about objects, hence the central concept underlying the Fox Query Language (FQL) is the *heap object*. 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 of objects returned by filters. Queries include a standard set of operations such as

Filter syntax followed by its description
`FSnapshot();` This filter represents all the objects in the heap snapshot.
`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.
`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.
`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.
`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.
`FUnion(<FILTER>, <FILTER>);` This filter selects the objects from those that pass at least one of the filters given. No duplicates are returned.
`FIntersection(<FILTER>, <FILTER>);` This filter selects the objects from those that pass both of the filters given.
`FMinus(<FILTER>, <FILTER>);` This filter selects the objects from those that pass the first filter but not the second one of the filters given.
`FKids(<FILTER>);` and `FParents(<FILTER>);` These filters select the objects from those who are kids (or parents) of the objects passing the filter given.

Table 1: Filters supported by the FQL

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.

Figure 5 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.

3.3 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 must be calculated by Fox. 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), `PNumberOfStaticIncomingReferences` (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 (Lengauer & Tarjan 1979), 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 elsewhere (Potanin, Noble & Biddle 2003), 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.

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.

Table 2: Standard queries supported by the FQL. Note that in case of the *per class* version of a query, the percentage of objects with respect to all the objects of the same class is going to be returned.

3.4 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 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. To enable a user to easily refer to the children of parents of objects returned by the filters, we introduced `FKids` and `FParents`. Table 1 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.

3.5 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 one 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 2 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 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 3.

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 themself, rather than letting Fox run in a batch mode. They include a help

<table>
<tr><th>Query Syntax Followed by its Description</th></tr>
<tr><td><code>QLoadHeapSnapshot("/path/to/the/heap/snapshot.hprof");</code>

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 (Sun Microsystems 2002<i>b</i>).</td></tr>
<tr><td><code>QListHeapSnapshots();</code>

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.</td></tr>
<tr><td><code>QSaveResults("/path/to/the/results/file.txt", [true | false]);</code>
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.</td></tr>
<tr><td><code>QClearResults();</code>
This query simply clears the window with the results. It is useful in combination with <code>QSaveResults</code> query.</td></tr>
</table>

Table 3: Control queries supported by the FQL

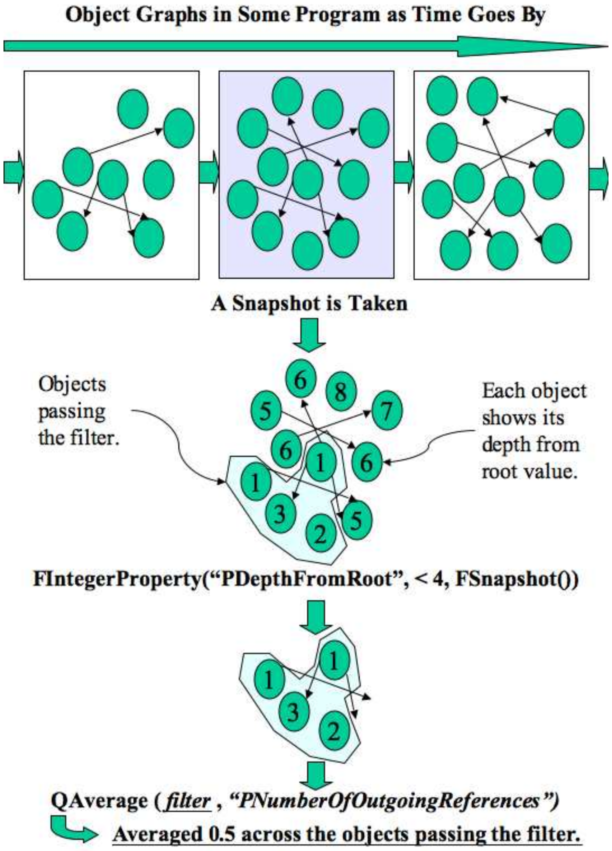


Figure 5: How queries, filters, and properties fit together

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 (Foote 2002) 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.

There is also a special query called `QProperties` that doesn't have to be used interactively. It is presented in table 4 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 using spreadsheets.

3.6 Extending the Fox

Finally, to conclude our discussion of the tool, we would like to briefly describe the way we designed Fox. After having had a good experience with Rabbit, we took an approach that led us to the simplest possible design that met 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 6 shows the essential classes and packages present inside Fox. There are few 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 `tools` 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.

4 Sample Applications

Here we present a number of simple examples that serve to illustrate the query language. They are not designed to go into detailed analysis, rather they just provide a feel for the query language. Comprehensive studies conducted using the Fox can be found in other publications (Potanin, Noble & Biddle 2003, Potanin, Noble, Frean & Biddle 2003).

We would like to start with a very simple example to introduce our 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:

```
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()));
```

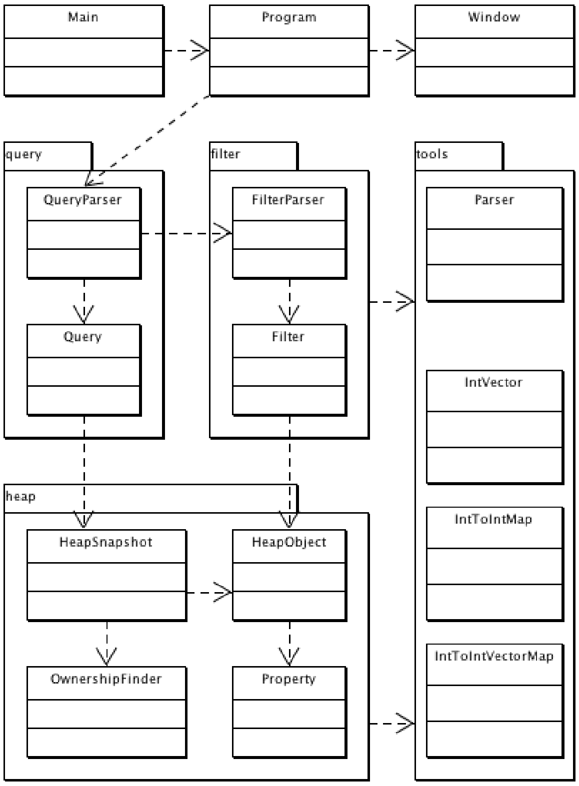


Figure 6: Class diagram of Fox

4.1 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 is 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:

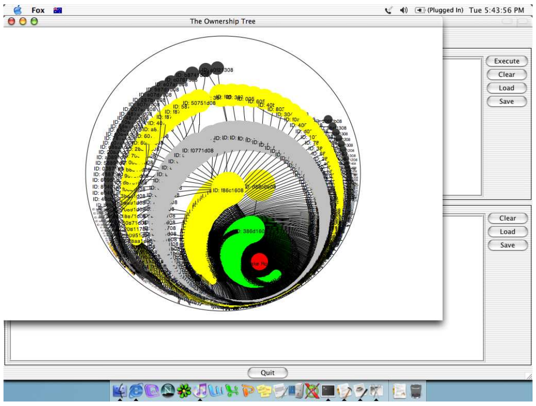


Figure 7: Visualising `HelloWorld` using Fish Eye View

```
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 likely that fields are pointed at by unrelated objects more often than non fields.

4.2 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.
        }
  }
 }
}
```

From our point of view, it is a rather simple program, but surprisingly, counting the number of objects in the heap snapshot taken at some time during the execution of the two loops:

```
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

Query Syntax Followed by its Description
`QHelp();` This query simply lists all the properties, filters, and queries supported by the tool with syntax and type information.
`QVisualiseTree([object ID]);` This query visualises a complete ownership tree using a fish eye view (VisualBeans.Com 2003). If the object ID is supplied, it only shows the objects below the given one in the tree, otherwise the whole tree is displayed.
`QHAT([port number]);` This query will start a server using the HAT library (Foote 2002) ability to allow a 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: Interactive queries supported by the FQL

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	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 a little bit	99,733

Table 5: Five snapshots of Fox taken as it processes a snapshot

Forte, containing around 350,000 objects. Running the `QVisualiseTree` query produced the fish eye view (VisualBeans.Com 2003) shown in figure 7.

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.

4.3 Checking the Fox

We obtained five dumps of Fox itself as it was working with the heap snapshot of the simple `HelloWorld` program described above. Table 5 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.

5 Conclusion

The contribution of this paper is to present the design and architecture of a new tool called Fox that can be used to analyse object graphs of running programs in detail. Fox is a general purpose tool utilising a query language in a style similar to Lencevicius' query-based debuggers (Lencevicius 2000) to enable a very detailed examination of heap snapshots — an approach utilised by a substantial number of research tools available (Foote 2002, AlphaWorks 2003, Mitchell & Sevitsky 2003).

Fox allows the user to examine all the information available about a program, not just parts related to the currently executing application, like DINO (Hill et al. 2000). It only requires heap snapshots and no additional information and is research-oriented, rather then being an invaluable debuggind aid, like HAT, Jinsight and Leakbot are (Foote 2002, AlphaWorks 2003, Mitchell & Sevitsky 2003). It can be (and was extended) to accomodate the questions that are of interest to a researcher currently using it and is available online (Potanin 2003). We plan to release it into the open source community in the near future.

We are still working on extending a number of aspects of the Fox, one of which is redesigning a query language to make it more flexible and allow queries that involve the code of the application in question. For example, we want to allow users to check if there exist two `Link` objects that have a field `next` pointing at the same object that is not `null`. Using the new query language, it will look like this:

```
#(l1 : Link, l2 : Link |
 (l1.next = l2.next) &
 (l1.next != null) )
```

Fox made it possible for us to conduct a study (Potanin, Noble & Biddle 2003) where we performed a variety of encapsulation, ownership and confinement metrics across a corpus of 60 heap snapshots taken from 35 programs. During the study, a question came up about a connectivity pattern typical of objects (if there is any). Fox allowed us to perform a small number of queries showing a distribution of incoming and outgoing references, which we found to follow a power law (Potanin, Noble, Frean & Biddle 2003), thus demonstrating that object-graphs have a scale-free structure found in networks around us such as the Web (Barabasi, Albert, Jeong & Bianconi 2000). We expect that Fox will continue to be useful in future research on object graph structure.

References

AlphaWorks, I. (2003), 'Jinsight', Available at: http://www.alphaworks.ibm.com/tech/jinsight/.

Barabasi, A.-L., Albert, R., Jeong, H. & Bianconi, G. (2000), 'Power-law distribution of the world wide web', *Science* **287**(2115a). In technical comments.

Foote, B. (2002), 'Heap analysis tool', http://java.sun.com/people/billf/heap/.

Grothoff, C., Palsberg, J. & Vitek, J. (2001), Encapsulating objects with confined types, *in* 'Proceedings of Conference on Object-Oriented Programming, Languages, and Applications', ACM Press.

Hill, T., Noble, J. & Potter, J. (2000), Scalable visualisations with ownership trees, *in* 'Proceedings of TOOLS Pacific 2000', IEEE CS Press, Sydney, Australia.

Lencevicius, R. (2000), *Advanced Debugging Methods*, Kluwer Academic Publishers.

Lengauer, T. & Tarjan, R. E. (1979), 'A fast algorithm for finding dominators in a flowgraph', *ACM Transactions on Programming Languages and Systems* **1**(1), 121–141.

Mitchell, N. & Sevitsky, G. (2003), Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications, *in* 'European Conference on Object-Oriented Programming (ECOOP)', Lecture Notes in Computer Science, Springer-Verlag.

Noble, J., Vitek, J. & Potter, J. (1998), Flexible alias protection, *in* 'European Conference on Object-Oriented Programming (ECOOP)', Lecture Notes in Computer Science, Springer-Verlag.

Potanin, A. (2003), 'The Fox project web page', Available at: `http://www.mcs.vuw.ac.nz/~alex/fox/`.

Potanin, A., Noble, J. & Biddle, R. (2003), 'Checking ownership and confinement', *Concurrency and Computation: Practice and Experience* . Accepted for Publication.

Potanin, A., Noble, J., Frean, M. & Biddle, R. (2003), 'Scale-free geometry in object-oriented programs', *Communications of the ACM* . Accepted for Publication.

Sun Microsystems (2002*a*), 'Java development kit', `http://java.sun.com/j2se/`.

Sun Microsystems (2002*b*), 'Java virtual machine profiler interface', `http://java.sun.com/j2se/1.4.1/docs/guide/jvmpi/index.html`.

VisualBeans.Com (2003), 'Fisheye bean', Available at: `http://www.visualbeans.com/FishEye/`.

Zeller, A. (2002), Isolating cause-effect chains from computer programs, *in* 'ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)'.

Zeller, A. (2003), 'Delta debugging', Available at: `http://www.st.cs.uni-sb.de/dd/`.

Zimmermann, T. & Zeller, A. (2001), Visualizing memory graphs, *in* 'Software Visualization', Vol. LNCS 2269 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 191–204.