Snapshot Query-Based Debugging

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 complex webs of objects and references between them, generally referred to as object graphs. These object graphs are difficult to design correctly and even more difficult to debug if incorrect. Unfortunately, very subtle bugs in object-oriented programs are directly caused by object graph topologies. Snapshot query-based debuggers let programmers examine object graph snapshots of programs in detail using a specially designed query language. This provides users with an ability to debug and examine their programs in great detail at the time when the memory snapshot is taken.

1 Introduction

Debugging programs is a task that software engineers face on a daily basis. Debugging of object-oriented programs is particularly hard because their behaviour is characterised by their object graphs — memory structures formed by the objects on the heap and the references between them. Analysis and debugging of object graphs has always been an active area of software engineering research [19, 3, 2, 6, 4, 11]. Such study can provide valuable insights into the real behaviour of a given program, sometimes different from the intended behaviour created by the programmer(s) who wrote the underlying classes.

Query-based debugging allows programmers to formulate queries about the state of the object graph of a program, or the state of a program at a given moment of time. Unfortunately, many of the query-based debugging tools require modifications to the program being debugged [4], or they require changes to the run time environment that executes a program being analysed [6].

In this paper we present a snapshot query-based debugger combining query-based debugging with close examination of heap snapshots. We access a complete state of the heap via the profiling interface and provide a query language that can be used to examine a heap snapshot in great detail. To demonstrate the feasibility of snapshot query-based debugging, we have implemented a proof-of-concept prototype tool, called Fox. It doesn't require any additional changes to the run time environment of a program, other than those already provided by modern virtual machines such as JVM or .NET CLR [14, 7].

Section 2 briefly surveys the area of object graph analysis and query-based debugging. Section 3 presents the query language that we developed and how it is designed to help users examine the heap snapshots to help with debugging or research. Section 4 presents a selection of examples, using the query language to debug and examine programs' memory. Section 5 surveys the related work. Finally, section 6 discusses future work. The detailed architecture and the design of the tool is described in a more detailed technical report [10].

2 Background

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

Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)
1530-0803/04 $ 20.00 © 2004 IEEE

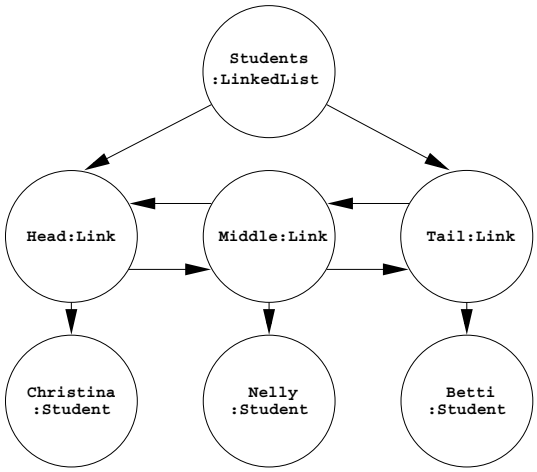

Figure 1. Simple object graph of a linked list

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` "Nelly") are accessed *uniquely* by only a single reference.

2.1 Heap Analysis

Most modern object-oriented languages, like Java or languages built on top of .NET, have managed memory to provide facilities such as garbage collection. This allows programmers to easily access any information about the contents of program's memory via special debugging interfaces. It is typically possible to dynamically receive events about object construction and destruction, or obtain a complete snapshot of the entire object graph — a *heap snapshot* — to analyse in detail.

Any modern debugger, for example DDD [16], can stop a program's execution and allow a programmer to examine the objects pointed at from the stack, or those stored in the heap. We can use breakpoints to stop the execution near the part of a program that we are interested in. Unfortunately, these tools typically provide us with an ability to examine the values of a particular variable or object, but not a coherent way to study general patterns or relationships within the heap.

An alternative approach is to examine a single heap snapshot, rather than monitoring a program as it executes. For example the Heap Analysis Tool (HAT) [1] is designed to allow a user to take a heap snapshot obtained from a running Java program and traverse the information about individual objects starting with the objects in the root set. Each object in the snapshot is represented using a web page that contains links to other objects that it points to.

2.2 Query-Based Debugging

Query-based debuggers [6,4] allow a user to examine the state of a program by formulating a query using a query language that the debugger can evaluate. Queries can be logical relationships between the values of variables (what is the average value of the field `size` among all the isntances of a `LinkedList`?), pre- and post- conditions around methods (is the method parameter `age` greater than zero?), relationships between pointers to objects (do these two pointers point at the same object), conditional breakpoints (if variable `n` at this point of a program is greater than five, then stop) etc.

Lencevicius's work in query-based debugging [6] allows a programmer to either dynamically (as the program is running) or statically (as the program's memory 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
Link* l1, l2;
// Relationship to verify.
l1.element != l2.element;
```

Lencevicius implemented a static query-based debugger in Self that evaluates a query by going through all the objects present inside the system as the query is executed. The dynamic query-based debugger accepted queries that were verified as a Java program was running and the user was notified in case when the program invalidated a query.

In his book, Lencevicius summarised the major kinds of queries that he encountered in his analysis. These included querying whether an object belongs to some collection or a number of collections, querying and comparing the values of fields of different objects (e.g. `size`), searching for duplicate pointers in linked data structures, and studying chains of references between objects.

This work clearly demonstrated the advantage of having a flexible query language that can be used to specify the inter-object relationships that a programmer wishes to verify. Unfortunately, the dynamic verification of queries requires a substantial extension of the Java Virtual Machine and the range of queries is limited so that dynamic monitoring would not slow down program's execution by a large factor. This makes this approach impractical for a detailed examination of the state of a large program.

The Object Constraints Language (OCL) C++ debugger by Hobatr and Malloy [4] allowed a user to specify the constraints on the functions using OCL to be validated as a C++ program runs. The constraints were translated into C++ code so that a program can print an error message every time a query expressed in OCL is invalidated. Again, this approach required transformation of a C++ program

and significant execution overhead, making this impractical for general-purpose debugging.

This and other approaches to debugging, for example conditional breakpoints that pause program's execution if a certain boolean expression is false, attempt to provide a programmer with a way to deal with complexity arising in modern programs that obscures the causes of errors.

3 Snapshot-Based Query-Based Debugging

In this paper we introduce a general purpose snapshot query-based debugger called Fox. Unlike other query-based debuggers, the main contribution of Fox is that it supports queries over standard heap snapshots. Basically, you can think of this approach as pausing a program and studying its state in great detail to look for causes of program behaviour using all the information available. This makes it instantly usable by anyone using Sun's Java Development Kit [14] and does not require any changes to the programs being examined.

The argument of this paper is that applying query-based debugging techniques to heap snapshots with instantaneous but complete information about the heap, allows access to information not available to standard debuggers but without the program transformations, virtual machine support, and execution overheads required by other query-based debuggers.

3.1 The Fox Query Language

The study of object graphs is about objects, hence the central concept underlying 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 sets of objects returned by filters. Queries include standard operations such as counting the objects or finding a minimum or a maximum value of a particular property, 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 interactive queries that are used in real time.

Figure 2 gives a visual example of how a typical query works together with filters and properties. As can be seen from the figure, Fox manipulates sets of objects before arriving at a final set that is supplied to a query being performed. In the rest of this section, we address the three fundamental concepts of FQL: properties, filters, and queries.

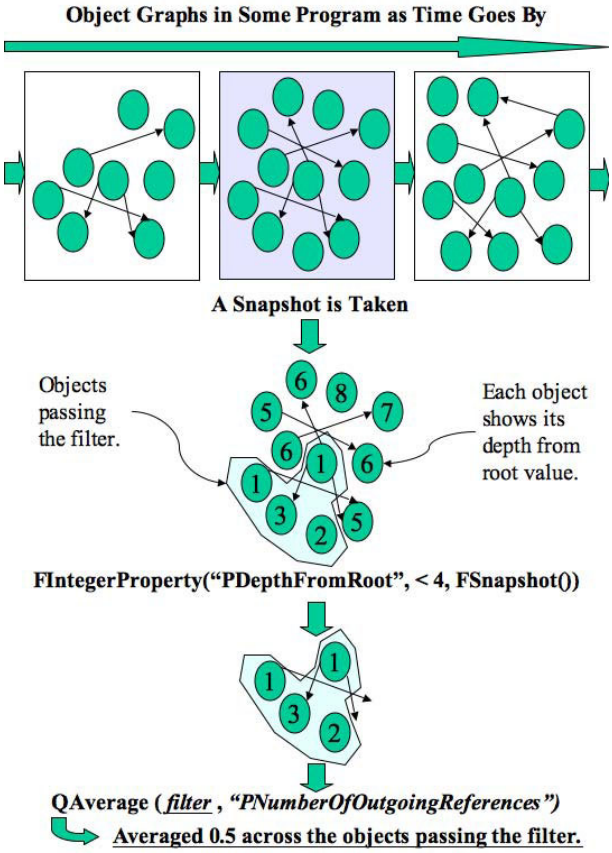

Figure 2. How queries, filters, and properties fit together

3.2 Properties

Most information about an object is recorded in its properties. Each property has a type and can be used in filters and queries as appropriate. In particular, the `pField` property used to access the values of fields of primitive type can be any of the following Java types: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, or `short`.

Some properties are calculated by Fox as the heap snapshot is loaded. These include a `pID` (object's unique ID) and `pClassName` (object's class name) that come from the heap itself as they are intrinsic to the objects when used by the garbage collector inside the Java Virtual Ma-

Filter Syntax
`fSnapshot();` This filter represents all the objects in the heap snapshot.
`fBooleanProperty("<PROPERTY>", [true | false], <FILTER>);` This selects objects from those that pass a given filter and have a given property being either true or false as given.
`fIntProperty("<PROPERTY>", [=, !=, >, <, >=, <=] [0-9]+, <FILTER>);` From objects passing a given filter, this selects those which have a property of the right type passing the comparison. In place of `Int` we can also use `Byte`, `Short`, or `Long`.
`fCharProperty("<PROPERTY>", [=, !=] <char symbol>, <FILTER>);` From objects passing a given filter, this selects those with a given character property passing the comparison.
`fDoubleProperty("<PROPERTY>", [>, <] [0-9]+[.][0-9]*, <FILTER>);` Returns objects which have a property passing a comparison given (`Double` can be replaced with `Float`).
`fStringProperty("<PROPERTY>", <user string>, <FILTER>);` From objects passing a given filter, this selects those with a given string property being equal to a given string.
`fUnion / fIntersection / fMinus (<FILTER>, <FILTER>);` Selects the objects from those passing two filters given and returns union, intersection, or difference of the two sets.
`fRefersTo(<FILTER>); fReferrers(<FILTER>);` These filters select objects that those passing the filter point to (refers to) or those that point to them (referrers).
`fField("<reference type field name>", <FILTER>);` This selects an object pointed at by a given field, or in case when it is an object array — all the objects that are in it.
`fOccurs([=, !=, >, <, >=, <=] [0-9]+, <FILTER>);` Returns a list of objects *where each appears only once* that occur a given number of times (e.g. > 1 - more than once).
`fTraverse(<FILTER: starting>, <FILTER: among>>;` Returns objects traversable from those passing the first filter if only those passing the second one are allowed in the paths.

Table 1. Filters supported by FQL

chine [14]. These are simply read from a heap snapshot file and recorded appropriately. Most other properties are calculated by Fox.

`pID` contains an integer ID uniquely identifying an object inside the heap. (This ID is not guaranteed to be retained by the virtual machine once the object was destroyed, thus its use for tracking an object through multiple heap snapshots should be restricted.)

`pClassName` contains a string with a full class name of an object.

`pNumberOfIncomingReferences` contains an integer that stores the number of objects that have a pointer to a current object.

`pNumberOfOutgoingReferences` contains an integer that stores the number of objects that the current object points to.

`pIsField` is a boolean value that is true if there is some object on the heap that stores a pointer to the current object in one of its fields.

`pField("<primitive type field name>")` this can be used to access the values of primitive type fields stored inside the current object. If the value is a primitive type, the property has that type, otherwise this will cause Fox to refuse to parse this property. To refer to the fields pointing to other objects, we can use an appropriate filter, described in the next subsection.

3.3 Filters

For some queries it is more useful to work with only part of the heap snapshot, thus FQL requires some filtering ability. We introduce the concept of *filters* that 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 FQL lies in `fBooleanProperty`, `fByteProperty`, `fCharProperty`, `fDoubleProperty`, `fFloatProperty`, `fIntProperty`, `fLongProperty`, and `fShortProperty` 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 objects that directly refer to the set of objects in question, or that the

Query Syntax

`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 FQL. Note that in case of the *per class* version of a query, the results for all the instances of the same class for each class will be presented.

current set of objects refers to, we introduced `fRefersTo` and `fReferrers`.

Filter `fField` allows us to refer to the objects that are pointed at by a particular field of an object or objects returned by the input filter. In the case when the field refers to an object array, all the objects in it are returned. Filter `fOccurs` allows the selection of objects that occur more than once in the list returned by the input filter.

Finally, filter `fTraverse` allows us to find the objects traversable from a given set of objects if we only examine objects which pass the second filter given as a parameter. It will start with object(s) returned by the first filter and traverse their children only if they are among the objects contained in the second filter. Notice that this is different from a query asking for reachable objects, that can take a long time on even typically sized heap snapshots. For example, it will ignore any outgoing reference that doesn't match the second filter, even if that path will eventually reach an object of a valid type. Specifying the second parameter to be `fSnapshot` makes this query return a valid set of reachable objects. An example that utilises this query is given in section 4.

Table 1 describes the filters supported by FQL. The reason behind a somewhat cumbersome syntax is the ease with which filters can be parsed and the way it allows a user to be sure that they are supplying values of the right type.

3.4 Queries

Queries are where all the work is coordinated. 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 a *per class* version, where the results are reported with respect to the instances of the same class only. For example, the average number of outgoing references across all objects can be two, while the average number of outgoing references across the objects of class `HashtableEntry[]` (array) can be a lot higher. The per class versions of queries return as many results as there are classes used in a heap snapshot.

Like most SQL implementations, FQL allows 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 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. 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`), 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 con-

Query Syntax
`qLoadHeapSnapshot("/path/to/the/heap/snapshot.hprof");` Loads a heap snapshot stored in a file given, which is obtained using the HPROF library [15].
`qSaveResults("/path/to/the/results/file.txt", [true | false]);` Saves the results to a file given; second parameter specifies whether this query is allowed to overwrite it.
`qClearResults();` Clears the window with the results; useful in combination with `QSaveResults` query.
`qHelp();` Lists all the properties, filters, and queries supported by the tool with syntax and type information.
`qHAT([port number]);` Starts a server using the HAT library [1] to allow the use of 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.
`qShow("<FILTER>");` This query is used to produce a brief, nicely formatted textual description of the objects passing the filter. It is only useful when there are only a few (less than a dozen) objects involved.

Table 3. Control and interactive queries supported by FQL

trols 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 themseves, rather than letting Fox run in a batch mode. They include a help command (`qHelp`), a query to traverse the memory graph of objects using HAT [1] that will serve them on a given port for the user to use any web browser to explore it (`qHAT`), and a command to show a user readable description of a single object (`qShow`). These are described in table 3.

There are also special queries `qProperties` and `qShow` presented in table 3. `qProperties` doesn't have to be used interactively: it 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. `qShow` is a simplified version of `qProperties` that can be used as a shortcut to display nicely formatted information about (hopefully few) objects returned by the filter.

4 Examples

In this section we would like to present some examples of using a query language to debug and examine a heap snapshot. Each example tries to briefly explain the thought process used to arrive at a particular query.

4.1 Selection Based on Field Values

This example reports the percentage of instances of an `Employee` class in a Java-based payroll application that have a value stored in the integer field `salary` greater than $\$50,000$ and the integer field `age` smaller than 20.

When building up any Fox query from scratch, we start by thinking about all the objects in the object graph (returned by `fSnapshot()` query). In this case, we first restrict our attention to those objects are instances of `Employee` class:

```
fStringProperty("pClassName",
   "Employee", fSnapshot())
```

Then we impose two further restrictions (age and salary) as follows:

```
fIntProperty("pField("age")", < 20,
   fIntProperty("pField("salary")",
      > 50000, ...))
```

Finally, we run a `qPercentagePerClass` query that will tell us what percentage of these `Employee` objects meet our restriction. The whole query is:

```
qPercentagePerClass("fIntProperty(
   "pField("age")", <20, fIntProperty(
   "pField("salary")", >50000,
      fStringProperty("pClassName",
                      "Employee",
                      fSnapshot()))");
```

4.2 Examining a Collection

This example reports the number of linked list `Links` that store an object of type `Student`. Again, we restrict our attention to the objects of class `Link` and then we calculate the set of objects constituted by those pointed at via the field `element` of class `Link`:

```
fField("element", fStringProperty(
   "pClassName", "Link", fSnapshot()))
```

Among these objects, we select those that are instances of class `Student` and we count them:

```
qCount("fStringProperty("pClassName",
   "Student", fField("element",
      fStringProperty("pClassName",
         "Link", fSnapshot()))");
```

4.3 Searching for Linked Data Structure Bugs

This example finds the number of objects pointed at by the same linked list `Links` via the field `next`. This can be used to detect errors in a program that cause broken linked lists. First we select those objects pointed by a field `next` among the instances of class `Link`:

```
fField("next", fStringProperty(
   "pClassName", "Link", fSnapshot()))
```

Then, we use a filter `fOccurs` to count only those objects that occure more than once in our list of objects pointed to by the field `next`:

```
qCount("fOccurs(> 1, fField("next",
   fStringProperty("pClassName",
      "Link", fSnapshot()))");
```

4.4 Debugging a Data Structure

This example examines the field `size` of an instance of `BinaryTree` data structure implemented by using a `BinaryNode` class (the topmost node stored in a field `root` of `BinaryTree`) to represent its nodes and checks if it indeed matches the number of nodes in the data structure. It relies heavily of `fTraverse` filter.

We can first look up the values of a field `size` for the instances of `BinaryTree` as follows:

```
qProperties("fStringProperty(
   "pClassName", "BinaryTree",
      fSnapshot())",
         "pID", "pField("size")");
```

Then, for each `ID` that we get we can run a query that counts the number of `BinaryNode` instances that a given `BinaryTree` instance can reach. The results an be compared with the value of the field `size` by exporting them to a spreadsheet program.

Before we can use `fTraverse` filter, we need to find the root object of the binary tree, stored in the `root` field:

```
fField("root", fIntProperty("pID", ID,
   fSnapshot()))
```

Then we start at this object, traverse downwards through objects of type `BinaryNode` and count them:

```
qCount("fTraverse(...), fStringProperty(
   "pClassName", "BinaryNode",
      fSnapshot()))");
```

4.5 Discovering a Singleton Bug

This example assumes that we have a class that is only allowed to have a single instance present inside any running program. The following is one of many singleton classes in `javac` compiler source:

```
public class Resolve {
  public static Resolve
  instance(Context context) { ... }

  ...
}
```

One of the possibilities that a Snapshot-based Query-Based Debugger offers is verifying that during the execution of a program there is only one instance of such class present.

```
fStringProperty{"pClassName",
   "Resolve", fSnapshot())
```

The above gives us access to all the instances of class `Resolve` that can be counted to make sure that there is only one of them.

```
qCount(fStringProperty{"pClassName",
   "Resolve", fSnapshot())
```

5 Related Work

Object graphs have been examined using a variety of methods which include: using traditional debuggers [16]; viewing the contents of the heap in detail [1]; examining a series of consecutive heap snapshots of a program run to help debug errors automatically [17]; and analysing heap snapshots in detail to try and find places where there is a high probability of memory leaks [8] or profiling execution behaviour in detail [5].

5.1 HOWCOME and Delta Debugging

HOWCOME is a cause-effect gap detector written by Zeller [17]. It constitutes a part of the work on the Delta Debugging Project [18].

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.

5.2 Leakbot and Jinsight

Leakbot [8] was developed as part of the Jinsight project at IBM Research and it provides a user with a 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.

Jinsight [5] 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.

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

5.3 Other Approaches

Kacheck/J [2] 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.

DINO is a tool devloped by Trent Hill [3] that visualised an ownership tree — a structure derived from the object graph used in the aliasing research. One of the issues encountered during the development of DINO was 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.

6 Discussion and Future Work

To summarise the main idea, although the best approach would be to dynamically analyse the behaviour of a program in real time, this is not feasible in current computing environments. Statically analysing heap snapshots in detail allows us to have access to complete information about a program's state. Although this approach is optimistic — the state of a program changes, as it moves on from the time the snapshot is taken — in practice we and other researchers working with snapshots found that generally the information about the inter-object relationships is stable.

From our experience with Fox, the main drawback is the syntax of FQL. We are currently designing a much more user-friendly version that will replace it. For example to find the number of instances `Link l` where `l.next.prev != l`, we will form the following query:

```
fox> #(Link l | l.next.prev != l);
```

The timing of when a heap snapshot is taken directly affects whether we are able to "catch" a particular state that we want to examine in detail. If we don't have access to the program's source code, we can can only produce heap snapshots at the point of time when a user thinks is reasonable. It is more useful to insert a line of code in the Java source that tells a virtual machine to produce a heap snapshot. This capability was not implemented at the time of

writing, because we chose instead to insert a simple call to `System.sleep()` to make it easy to dump a heap snapshot manually when the program pauses.

We are also planning to use BeanShell [9] to allow a user to execute methods upon the objects inside the heap snapshot. To implement this, we can dynamically recreate the program with BeanShell using the information stored in a heap snapshot.

Finally, we plan to release Fox into the open source community once it is ready. So far, a small web page was set up at: http://www.mcs.vuw.ac.nz/~alex/fox/ that contains a link to download the latest stable version. We welcome any requests from people who would like to obtain the source.

7 Conclusion

In this paper we describe a novel approach to debugging object-oriented programs. Snapshot query-based debuggers provide a query language that can be used to study snapshots of running programs in detail. This approach is simple and does not require any modifications to the run time environment of programs being examined. It is possible to use such a debugger on any project without any interference with the development practice. Fox has already proven useful in a number of projects, including the development of Fox itself and a Java-based compiler project Oh! Gee! Java! [12, 13].

Acknowledgements

This work is supported in part by the Royal Society of New Zealand Marsden Fund.

References

[1] B. Foote. Heap analysis tool. http://java.sun.com/people/billf/heap/, 2002.

[2] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications*. ACM Press, 2001.

[3] T. Hill, J. Noble, and J. Potter. Scalable visualisations with ownership trees. In *Proceedings of TOOLS Pacific 2000*, Sydney, Australia, 2000. IEEE CS Press.

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

[5] IBM AlphaWorks. Jinsight. Available at: http://www.alphaworks.ibm.com/tech/jinsight/, 2003.

[6] R. Lencevicius. *Advanced Debugging Methods*. Kluwer Academic Publishers, August 2000.

[7] Microsoft. The .NET Common Language Runtime. http://msdn.microsoft.com/net/, 2003.

[8] N. Mitchell and G. Sevitsky. 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, July 2003.

[9] P. Niemeyer and D. Leuck. BeanShell — Lightweight Scripting for Java. http://www.beanshell.org/, 2003.

[10] A. Potanin, J. Noble, and R. Biddle. The fox — a tool for object graph analysis. Technical Report CS-TR-03/15, School of Mathematical and Computing Sciences, Victoria University of Wellington, 2003.

[11] A. Potanin, J. Noble, and R. Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 2004. Accepted for Publication.

[12] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Featherweight generic confinement. In *Foundations of Object-oriented Programming (FOOL11)*, Venice, Italy, January 2004.

[13] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004. Submitted for publication.

[14] Sun Microsystems. Java development kit. http://java.sun.com/j2se/, 2002.

[15] Sun Microsystems. Java virtual machine profiler interface. http://java.sun.com/j2se/1.4.1/docs/guide/jvmpi/, 2002.

[16] A. Zeller. Data display debugger. http://www.gnu.org/software/ddd/.

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

[18] A. Zeller. Delta debugging. http://www.st.cs.uni-sb.de/dd/, 2003.

[19] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Software Visualization*, volume LNCS 2269 of *Lecture Notes in Computer Science*, pages 191–204. Springer-Verlag, May 2001.