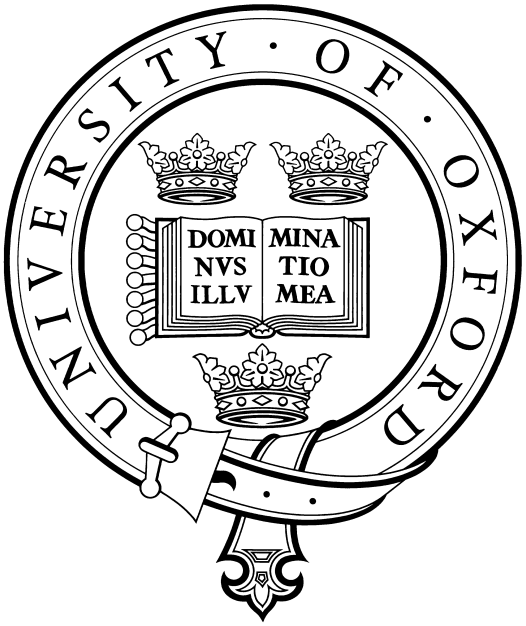
Static Querying Of Source-code Including Control-flow Analysis

Fintan Fairmichael

Under the supervision of *Prof. Oege de Moor*



Computing Laboratory

University Of Oxford

A thesis submitted for the degree of

MSc Computer Science

September 2006

Abstact

Tools designed to aid the software development process are in widespread use by millions of programmers throughout the world. Each tool is designed to help the programmer in some way, by making some programming tasks more straightforward, allowing software to be developed faster, or by generally trying to improve the quality of the software produced. Code querying programs are a particular type of these tools, aiming to determine properties contained in the code that are not immediately apparent. In a way, they aim to answer some question that a programmer might have about some given code. This information can then be used to improve the programmer's understanding of the program, or in some other way to improve the software — for example, fixing a bug that was detected. This paper examines the extension of an existing code querying system, CodeQuest, allowing it to gather and utilise a finer-grained level of detail from the input source code. CodeQuest provides a flexible, easy to use and scalable querying mechanism, utilising Datalog for query specification. The development of numerous useful queries, including a points-to analysis for Java, for this extended system are described and performance measurements given. The implementation of this extension of CodeQuest is also discussed.

Contents

Chapter 1

Introduction

1.1 Motivation

In recent times software developers have had a plethora of tools at their disposal to aid in the task of writing software. There is a widespread use of Integrated Development Environments (IDEs), providing a programmer with a cohesive suite of built-in functions at their immediate disposal.

The aim of all of these tools is to improve the software development process. This could be by making some programming tasks more straightforward, allowing software to be developed faster, or by generally trying to improve the quality of the software produced.

We define a code query to be an automated task, the results of which will inform the querier of properties contained in the code that were not immediately apparent. This could be anything from a simple search for text in the source code to a complex query that requires an understanding of the code's semantics. A code querying system is thus one which allows a user to perform these tasks. The aim of such a tool is largely to improve software quality, although it will likely also have other benefits.

Software queries become increasingly useful as the size, and complexity, of a body of software increases. It is therefore of the utmost importance that a querying system can scale to handle queries across large bodies of code efficiently. The desire to produce a general-purpose querying system that was expressive, easy to use, functional, and scalable led to the development of the CodeQuest[7] query tool.

The CodeQuest system was largely inspired by the JQuery[23] query-based source code browser. JQuery successfully offered an intuitive and flexible querying environment, allowing users to make use of built-in queries as well as write their own. However, due to the design of the JQuery tool, it is unable to scale to very large programs as it requires a subsection of the collected facts to be stored in memory at all times. This was one of the leading motivations for developing the CodeQuest system, that aimed to overcome these difficulties by building on top of a relational database to achieve scalability.

CodeQuest was shown to scale well to large sets of input code, allowing it to efficiently execute a wide range of queries over input programs of varying sizes. It also allowed users to create their own queries, or to modify existing ones, giving great flexibility in the queries that could be executed. Queries were written in a form of Datalog, giving users a desirable

balance of conciseness and expressivity.

The initial development of CodeQuest aimed to mimic the functionality provided by JQuery. Its fact-gathering mechanism therefore did not collect and store all information that was available from the source. For example, no information about assignments or the control-flow of a program was collected. For this reason, a variety of queries could not be represented in the system. This thesis examines the extension of the CodeQuest system to allow it to collect more detailed information about the input, as well as using this additional information in potential queries.

1.2 Contributions

The contributions made by this project can be summarised as follows:

- The extension of the CodeQuest fact-gathering system to collect information from Java source at a more fine-grained level, to the level of statements, expressions and all subexpressions.
- The generation of a control-flow graph for a given body of Java code, that is also constructed when input source code is parsed.
- Numerous useful Datalog queries/predicates that can be ran independently, or combined into a more complex query, that allow us to determine useful properties about the given input program. Included in this is the development of a points-to analysis for Java programs.
- The examination of the performance of these new types of queries for input programs of varying sizes, including the comparison of our points-to analysis against a state-of-the-art implementation.

1.3 Outline

We start in Chapter 2 with an introduction to static source-code querying, looking at some of its potential uses. We then examine in more detail some of the prior work that had the greatest influence on this project.

In Chapter 3 we briefly introduce Datalog as a query language, looking especially at the CodeQuest implementation of Datalog. We then detail the facts that are stored to facilitate our queries, and describe the development of several useful queries that can be executed on the newly extended CodeQuest system.

Chapter 4 describes the CodeQuest implementation. We examine the architecture and the integration of CodeQuest into a leading IDE as a plug-in for the Eclipse Platform. We then look in detail at the method of collecting the required information from source code, and examine some of the design decisions taken during the development process.

In Chapter 5 we illustrate the performance of CodeQuest by outlining a number of tests that were run on our system and, where appropriate, compare the performance with another leading tool.

In Chapter 6 we conclude on the work completed. We also outline possible future work that could form the next steps in the development of the CodeQuest system.

Chapter 2

Static Source-code Querying

This work falls under the category of static source-code querying. The information available to these queries is only that which is available at compilation time. Furthermore, the information collected for use in these queries is collected at the source-code level, and not from any other intermediate representations of code. We do this as we wish to be able to link results from queries back to the relevant locations in the source-code. For example, if we wish to run a query to identify methods with certain properties, we want our results to be able to exactly identify those methods in the original source-code.

There are numerous reasons why one would want to perform querying on a set of source code. Some of the most common motivations are listed below:

- Source-code Comprehension.
  As a body of code becomes larger, it will undoubtedly become more complex. Performing certain queries might give a programmer quicker access to information about a program, or information they might not be able to ascertain themselves at all. This is especially true for large bodies of code. Thus, a query can be used to aid a programmer's understanding of the code.

- Bug Detection.
  Being able to automatically detect potential bugs in software is a hugely useful capability. Automated bug detection will likely be very cheap, and might even find bugs that would otherwise be missed. Certain bug detection methods can be formulated as queries.

- Software Metrics.
  Software metrics are used to place a numerical measurement on certain interesting properties of a piece of software. For example, a measurement of the coupling, cohesion or cyclomatic complexity of a given program.

The most basic form of static querying is to do a simple text search. There are numerous tools available to perform such a task, such as `grep` or almost any rich text editor. Simple text searching is very fast and requires no prior processing of the code. The power of a text-based search can be enhanced by the use of regular expressions, but this technique is still

very limited as it does not utilise any of the semantics of the target language. Therefore, a purely text-based search cannot represent all of the queries that we might wish to perform.

Modern programming Integrated Development Environments (IDEs) tend to offer the user a variety of queries designed to aid a programmer. For example, they might offer the ability to detect all locations in code where a certain variable or method is referenced. Whilst these implementations tend to be well integrated into the development environment and also quite efficient, they offer little in the way of flexibility. Users cannot adapt queries to their individual needs, nor create their own, without programmatically altering the IDE in question or adding a plug-in to it (assuming this is possible).

There is therefore the desire to have a facility for programmers to create their own queries. Such a facility must be easy to use, efficient, scalable, expressive enough to create complex queries, and provide results in a manner that is useful to the programmer. It would be a great advantage if queries can be constructed modularly, allowing them to use simpler parts in more complex queries. It would also be advantageous to have a library of built-in queries that can either be directly used by a programmer, or combined into a custom query of their own.

There are many systems for code querying described in the literature (e.g. [29, 16, 6, 8]). However, many lack the scalability, generality, or ease of use that we seek. In this thesis we are discussing the extension of a general-purpose querying system that has been shown to have many of these properties, and this work will allow it to perform queries that utilise fine-grained details of source code and control-flow information. For a full background on the research and systems that led to the development of the CodeQuest system, see the original thesis on the tool[17].

2.1 Prior Work

Here we present some of the code querying/analysis systems that were the most influential on the development of this project.

2.1.1 bddbddb

`bddbddb`[3] is an implementation of a deductive database based on binary decision diagrams (BDDs). The tool implements a version of the declarative programming language Datalog, for manipulating relations stored in the database. `bddbddb` restricts itself to the use of *safe* Datalog programs — stratified programs for which minimal solutions always exist.

For running queries over programs a user must provide the relations to be used in the query, as the `bddbddb` tool does not provide this functionality directly. Instead, the relations can be generated by using the Joeq virtual machine and compiler infrastructure[22, 37]. Although not its main functionality Joeq provides a method for inputting compiled Java files and, making use of the Byte Code Engineering Library (BCEL)[2], generates as output a set of BDD relations that will enable the `bddbddb` tool to run analyses on the input program. Since the information collected is done so at the byte-code level, utilising `bddbddb` in this way is not strictly source-code querying. However, the close relationship between Java source-code and byte-code and the fact that the tool could be given input relations gleaned from actual source-code, means that we will still consider it in this category.

The `bddbddb` tool can be used to run a variety of program analyses, including: context-insensitive and context-sensitive pointer analysis, external lock analysis and SQL injection detection. It is not a general querying tool as such, being designed to perform these types of complex analyses. Empirical results have shown that the formulation of these queries in Datalog and their execution using the `bddbddb` tool allows for an efficient and scalable solution[38, 40].

2.1.2 JQuery

JQuery[23] is a query-based source code browser, implemented as a plug-in for the Eclipse Platform. It is implemented on top of an expressive logic query language, TyRuBa[34]. Its main aim is to combine the advantages of a hierarchical browser with the flexibility of a query tool[21].

The JQuery Eclipse plug-in allows users to specify bodies of code upon which to perform queries by utilising the Eclipse Platform's built-in tools for creating and storing "working sets". This allows users to easily select the source files they wish to include when they execute a query. Once a user has selected the body of code they wish to run queries on, the JQuery tool sets about collecting the necessary facts from the chosen code. This fact-gathering process operates on facts obtained from Abstract Syntax Trees (ASTs) generated by the Eclipse Java Development Tooling plug-in. The information that is gathered is stored in a logical database. So long as the user does not change the working set upon which they wish to run queries, the process of collecting the information into the database only needs to be performed fully once, as the JQuery tool will incrementally update the stored facts on the fly when it is informed of source code changes by the Eclipse Platform[28].

The JQuery implementation provides a fully-functional user interface for utilising the tool, making it more practical for use as a code browsing/querying system. Queries are written in a Prolog-like language, and a dialog box is provided to enter queries. An input query is parsed and, if syntactically correct, the variables used in the query are determined. The user is then asked to order the variables in the manner in which they want the results to be shown.

JQuery provides a number of pre-defined queries that allow a user to immediately start making use of the tool. Numerous built-in predicates can be utilised when writing queries, tailoring queries to suit their purpose. Additionally, advanced users can add to the pre-defined queries with queries of their own. This provides a great deal of flexibility to the type of query that can be executed on the system. As a result, a wide range of complex queries can be represented and executed.

Each component of a result is a particular item in the source code — for example: a package, type, method or field. The results of executed queries are shown in a hierarchical manner, with each result represented by a tree. The top-level node for a result will be the item that matched the variable listed first in the ordering, for the given result. Expanding the top-level node will reveal the variable listed next, again for the given result, and so forth. The results pane makes use of the built-in Eclipse view for showing hierarchical program structure. Thus, experienced users of the Eclipse Platform will be accustomed to browsing code elements in this manner.

It is possible to write recursive queries for use on the JQuery system by editing JQuery

configuration files[36]. Users wishing to write queries of this form must then give the definition of recursive predicates in TyRuBa. This detracts from the ease of use of the tool for writing more complex queries as users wishing to write such queries must also learn how to formulate queries in TyRuBa. Furthermore, the specification of predicates in TyRuBa is more complex, as type and binding information of the input variables must be supplied.

JQuery's parsing system is reasonably efficient, especially due to the incremental update process utilised after code has been parsed for the first time. However, a certain subset of the factbase collected from the code must be stored in memory at all times. This means that given an arbitrarily large set of input the tool will run out of memory and crash. This obviously limits the ability of the tool to scale to large programs.

2.1.3 CodeQuest

The functionality and versatility offered by the JQuery tool was largely the inspiration of the CodeQuest[7] system. CodeQuest aimed to provide the ability to express the same set of queries, whilst providing a framework that allowed the tool to scale to very large sets of input code. Removing the need to write queries in more complex forms, such as in TyRuBa, was also desirable. In essence, CodeQuest aimed to provide a querying environment that gave the same expressivity in query language, whilst also achieving scalability, and without requiring complex query annotations[18].

CodeQuest's querying is provided through a version of Datalog, a subset of the logical programming language Prolog. Similarly to `bddbddb`, it places certain stratification restrictions on the use of negation and recursion. Still, recursive queries are allowed, with no extra annotations needed to specify these. This allows CodeQuest to present a query language that is easy to use, expressive enough to write code queries, but it can also provide a scalable implementation by building on top of a relational database[19].

We examine querying using CodeQuest further in Chapter 3, and the implementation in Chapter 4. Further information can also be found by consulting the original thesis on the system's development[17].

Chapter 3

Querying Using Datalog

In this chapter we justify our choice of Datalog as a query language and give a brief introduction to the use of Datalog in CodeQuest. We then detail the information that we store in the database for use in our queries, and finally we outline the development of a number of useful queries that can be run on our system.

3.1 Datalog As A Query Language

It has been shown that we can succinctly express many program queries and analyses using general purpose logical programming languages such as Prolog and Datalog[9]. However, the typical queries that we are writing for use in the CodeQuest system do not require the full power of Prolog. Furthermore, Prolog requires that all facts be kept in memory, which reduces scalability. Datalog is syntactically a subset of Prolog, and does not allow data structures such as lists. Datalog does not require all facts to be kept in memory, and therefore presents a more scalable solution. We therefore choose Datalog to be our query language as it forms the right balance between expressivity and scalability.

3.1.1 CodeQuest Datalog Implementation

Here is a brief outline of the syntax and semantics used in Datalog predicates for our system. Readers already familiar with Datalog may skim through this section. This is an informal summary and far from a complete reference. It is intended solely to give a gentle introduction to some of the nuances in order to aid the reader's understanding of the rest of this chapter. For further information on the CodeQuest Datalog implementation see the Master's thesis by the tool's original author[17].

Firstly, we introduce some of the basic Datalog terminology. Predicates are the fundamental unit in Datalog. They consist of a head, or a name, and a number of arguments. For example: `old(X)`. We can determine whether a given predicate matches for certain arguments by consulting our stored database of facts. In the CodeQuest system the name of a predicate maps directly to the name of a database table. We state that a predicate `p(x1, x2, ... , xn)`, where `x1, x2, ..., xn` are all values, is true if there exists a row in table `p` containing the values `x1, x2, ... , xn` in that exact order.

Non-constant arguments used in predicates are considered to be variables. In the CodeQuest Datalog implementation strings contained within inverted commas (e.g. '`static`') and numeric values (e.g. 3) are constants, everything else is a variable. When we provide a variable as an argument to a predicate (and the argument has no other bindings) then the variable can take any value that satisfies the predicate. For example, in the predicate `p(X)` the variable `X` can take the value stored in any row in the table `p` in the database.

A rule, or clause, defines a new predicate through the use of already defined predicates. For example: `a(X) :- b(X).` means that predicate `a` matches with `X` if predicate `b` matches with `X`. A rule/clause is terminated by a full-stop. The common style is to give variables names using upper-case characters. We will continue this style, but it is not required for the CodeQuest implementation of Datalog.

We can add logical operators to our definitions of predicates. Commas (,) are used to mean a logical AND. For instance: `a(X) :- b(X),c(X).` means that predicate a matches `X` when both predicate b and predicate c match `X`. Similarly, a semicolon (;) stands for a logical OR. In which case `a(X) :- b(X) ; c(X).` means that predicate `a` matches `X` when either predicate `b` or predicate `c` match `X`.

Multiple definitions of the same predicate are allowed, forming a disjunctive definition of the predicate. We often refer to the constituent definitions of a single rule as "rule parts". As an example:

```
a(X) :- b(X).
a(X) :- c(X).
```

Is a logically equivalent definition of the predicate `a` as the below:

```
a(X) :- p(X) ; q(X).
```

The underscore character (_) is used as an argument to indicate "don't cares". These are used in predicates of multiple arguments to indicate that we will allow any value here.

We can negate predicates by encasing them in a `NOT(...)`, for example `NOT(old(X))` will match when `X` does not match in the predicate `old`. Due to restrictions on the use of negation however, variables used in a negated predicates must also be bound by another non-negated predicate in the same definition. For example, the following is not allowed:

```
a(X) :- NOT(b(X)).
```

But the below is fine:

```
a(X) :- NOT(b(X)),c(X).
```

We may also require that two items are identical, using the built-in `equals` predicate, of two arguments.

The tables stored in the database of our CodeQuest system form a set of immediate predicates for use in our Datalog queries. We then use these in new rule definitions to create more complex queries. This approach allows us to create sophisticated queries that are concise but whose meaning can still be readily understood.

3.2 Facts Stored For Use In Queries

Here we present the information that is now stored in the database for use in queries.

Note that where appropriate the location of an item in the source code is also stored. We simply state that a predicate stores its "position in source", but by this we mean that it stores the identifier of the compilation unit containing it, and the starting position and length of the characters representing it.

Table 3.1 details the predicates that were originally in the CodeQuest system, and remain unmodified.

Table 3.1: Original Predicates

Predicate Name	**Information Stored**
cus	Compilation unit identifier, name and path.
packages	Package identifier and name.
primitives	Primitive identifier and name.
classes	Class identifier, name and position in source.
interfaces	Interface identifier, name and position in source.
methods	Method identifier, name, signature and position in source.
constructors	Constructor identifier, name, signature and position in source.
modifiers	Modifier identifier and name (e.g. `static`).
params	Parameter identifier, name, type and position in source.
fields	Field identifier, name, type and position in source.
hasChild	Parent identifier and child identifier. For example, for type `T` and method `M` `hasChild(T,M)` means that `T` declares method `M`.
extends	Super-class identifier and sub-class identifier.
implements	Interface identifier and the implementing type's identifier.
hasModifier	Item's identifier and the identifier of the modifier it has.
returns	Method identifier and the identifier for the type it returns.
reads	Method identifier and the identifier of field read inside the method.
writes	Method identifier and the identifier of field written-to inside the method.
fullids	Stores a pair consisting of an item's numeric identifier and string identifier.

It was necessary to change a few of the original predicates. The changes were largely to add a unique identifier to each relation. This was necessary as we wish to have a single unique feature, an identifier, for all points in the control-flow graph. For example, method-calls originally did not have a single unique identifier stored. We add a unique identifier, as method-calls are unique points on a control-flow graph. Table 3.2 details these slightly modified predicates.

Table 3.2: Modified Predicates

Predicate Name	**Information Stored**
makesMethodCall	Method-call identifier, calling method identifier, called method identifier, expression identifier (or 0 if none), expression type (if no expression, the type that makes the method-call) and position in source.
makesThisCall	This-call identifier, calling method identifier, called constructor identifier and position in source.
makesSuperCall	Super-call identifier, calling method identifier, called method identifier and position in source..
makesConstructorCall	Constructor-call identifier, calling method identifier, called constructor identifier and position in source..

We now examine the predicates that were added to the system to store the extra information required. Table 3.3 outlines the new predicates.

Table 3.3: New Predicates

Predicate Name	**Information Stored**
controlFlowPoint	Stores the identifier of every point that appears in our control-flow graph.
controlFlowEdge	Stores pairs of identifiers representing an edge from the first to the second in our control-flow graph.
localVariable	Previously we were not concerned with local variables. We now store the local variable's identifier, name, type, control-flow point where it is declared and the position in source it is declared.
parameterExpression	An expression that is passed as an argument for a method-call. We store the identifier for the method-call, the expression's identifier and the index of the argument (e.g. the 3rd argument).
returnExpression	This relation identifies a return statement and the expression returned (if any). We store an identifier for the returns expression, an identifier for the method it is returning from and an identifier for the expression returned.
assignment	Represents an assignment that occurs in the source. Stores an identifier for the assignment, the type being assigned to, an identifier for the left-hand side expression, an identifier for the right-hand side expression and the position of the assignment in source.
inlineConditional	Represents an inline conditional expression in the source. Stores an identifier for the conditional, an identifier for the boolean expression, an identifier for the then expression and an identifier for the else expression.
variableAccess	Represents a variable access in the source. Stores an identifier for the control-flow point at which it occurs, an identifier for the variable accessed and the position in source it occurs.

Table 3.3: New Predicates (Continued)

Predicate Name	**Information Stored**
fieldAccess	Represents a field access in the source. Stores an identifier for the control-flow point at which it occurs, an identifier for the field accessed, and identifier for the expression (or 0 if none) and the position in source it occurs.
nullLiteral	Identifier for a point in the control-flow graph that is a null literal (i.e. `null`). Its position in source is also stored.
stringLiteral	Identifier for a point in the control-flow graph that is a string literal (e.g. `"st"`). Its position in source is also stored.
castExpression	Represents a class-cast operation in the source. Stores the identifier for the control-flow point, an identifier for the expression that is cast, and its position in source.
thisExpression	Represents a use of `this` in the source. Stores the identifier for the control-flow point, the containing method, and its position in source.
arrayVar	Stores the identifier of a variable that is an array type.
arrayCreation	Represents a control-flow point that creates an array. It can be either standard array creation (e.g. `new String[5]`), standard array creation with initialisers (e.g. `new String[] {"one","two"}`), or simply an array initialiser (e.g. `{"one","two"}`). Stores the control-flow point identifier, and the array type that is created.
arrayAccess	Represents an access to an array object, for example `a[6]`. Stores the control-flow point identifier and the array type expression that is being accessed.
arrayInits	Represents an array initialiser that is given as part of a standard array creation. Stores the identifier of the array creation it is initialising, and the identifier of the initialiser expression.
methodCallHolder	Stores an identifier and the identifier for the method-call.
fieldAccessHolder	Stores an identifier and the identifier for the field access.

3.3 Example Queries

In this section we present several example queries, giving some detail on how they were developed, their function and their potential applications. Some of these make use of the control-flow information that is now stored in the database, and others do not, but all of the queries given here rely heavily on the extra information now collected and stored as a result of this work.

Note that we make every effort to explain the Datalog code used, however, some details may be omitted. For the full Datalog code for each of these queries, see Appendix A.

3.3.1 Possible Methods Called

This query aims to match a method call invocation with the possible implementations of the called method that might be executed. To correctly explain the desired behaviour for this query, consider the two lines of Java code given below:

```
A a = getAnA();
a.m();
```

We are performing two simple actions: obtaining an instance of an `A` object through a call to an undefined method `getAnA()`, and invoking the method `m()` of no arguments upon this same object. We note the following facts:

- The object returned by the call to `getAnA()` can return an instance of an `A` object, an instance of a subtype of `A`, or the null pointer.
- The actual implementation of the method `m()` that will be executed will depend on exactly what type the variable `a` stores at the time of the method invocation.

Since we are performing a static analysis of source code, it is desirable to find every implementation of the method `m()` that could potentially be called at this point. It is exactly this question that we aim to answer in this query.

Compilers make use of the results of this type of query in order to reduce the cost of dynamic method invocation. If we know at compile-time that for a particular method invocation only one method may be executed, then an optimising compiler can improve the performance of this method-call by replacing the invocation with a direct call to that method or by potentially inlining that method's body. Alternatively, if we can only reduce the number of potential methods that are invoked to a certain set of methods, but that set is small, we can avoid the computation required in performing a general run-time method lookup. Instead we can insert a "type-case" expression, implemented with a series of run-time class tests. Each branch from this test would be to a direct procedure call implementing that case or to an inlined method.

Essentially what we must do here is to examine the class hierarchy for the type upon which the method is invoked — we are performing a class hierarchy analysis. There are many reasons to wish to perform this type of query to aid optimisation; for a more complete discussion on optimising object-oriented languages using class hierarchy analysis see the technical report on the matter by Dean et al.[10].

Our first consideration is as to whether the type `A` actually implements the method `m()`. By this we mean that `A` declares a method `m()` that is not abstract. We know that the object stored in `a` is either of type `A`, or some subtype of `A`.

- If `A` implements `m()` then we know that the implementation of `m()` that is actually executed will either be that of `A`, or any subclass of `A`'s implementation that overrides `m()`.
- If `A` does not implement `m()` then we know that the implementation of `m()` that is actually executed is either that declared by the nearest supertype[1] of `A` or the

[1] By nearest supertype we mean the first supertype with this property that we encounter while climbing the type hierarchy. For instance, for any class in Java, java.lang.Object is the farthest supertype whilst a type's direct supertype is the closest.

implementation provided by any subclass of `A` that overrides `m()`.

Thankfully in our stored relation for a method call the identifier for the actual method called is either that for the method that the type declares (if the type declares it), or the method in the nearest supertype declaring it. However, we must be careful here, we must also ensure that the type declaring the overriden version is in fact a subtype of the type upon which the method was invoked.

Finally, we can give our definition for methodCallToPossibilities, as in Listing 3.1.

```
1 methodCallToPossibilities(MC,M) :-
2     methodCallTo(MC,M),NOT(hasStrModifier(M,'abstract')).
3 methodCallToPossibilities(MC,M) :-
4     methodCallTo(MC,M1),overrides(M,M1),
5     methodCallExpressionType(MC,T),hasSubtypePlus(T,S),
6     declaresMethod(S,M),NOT(hasStrModifier(M,'abstract')).
```

Listing 3.1: Datalog definition of methodCallToPossibilities.

3.3.2 Detecting Direct Method-calls

We mentioned in the previous section that detecting method-calls that can only result in a single method-body being executed is desirable. We illustrate this idea by developing a short query that allows us to detect such points.

We first introduce a predicate that matches method-calls that can result in two or more method-bodies being executed. We do this by matching on method-calls that can call at least two different methods that are not the same:

```
possiblyCallsTwoOrMore(MC) :-
    methodCallToPossibilities(MC,M1),
    methodCallToPossibilities(MC,M2),NOT(equals(M1,M2)).
```

A method-call will always possibly call at least one method, so the method-calls we are interested in are those that do not call two or more. We thus do simply that — match on method-calls that do not match on `possiblyCallTwoOrMore`, as given in Listing 3.2.

```
1 directMethodCallTo(MC,M)    :-
2     methodCallToPossibilities(MC,M),
3     NOT(possiblyCallsTwoOrMore(MC)).
```

Listing 3.2: Datalog definition of directMethodCallTo.

3.3.3 Follows

It is now desirable for us to have a general purpose query that can tell us if one point in the control-flow of a program can follow after another. For this, we are essentially following the control-flow paths that lead from the first point. We can visualise this by looking at the

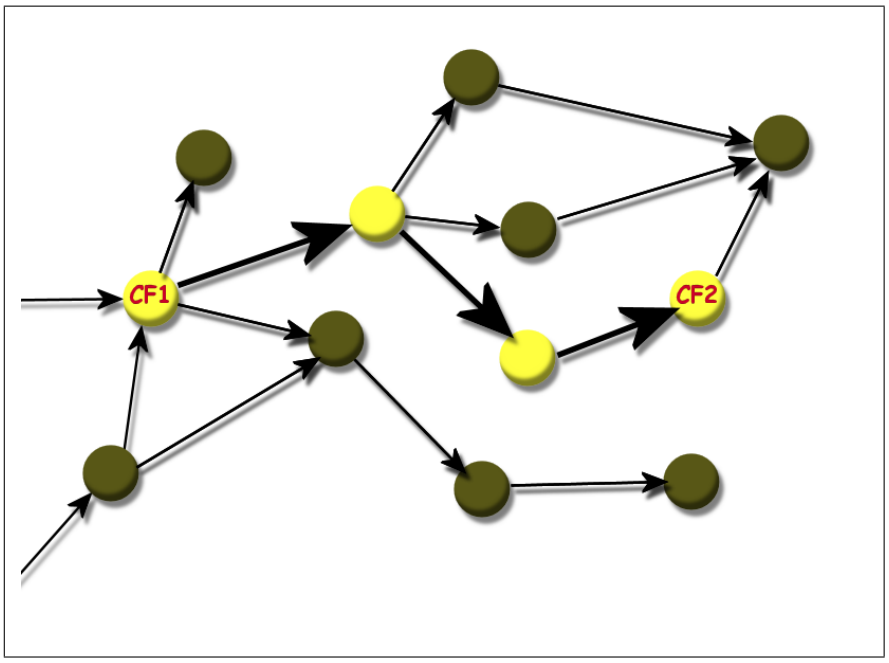


Figure 3.1: Graph diagram demonstrating follows

mock control-flow graph given in Figure 3.1. In this graph, the nodes are the control-flow points, and the arrowed lines are control-flow edges.

This query will match a pair of control-flow points, `(CF1,CF2)`, if there is a path in the control flow graph that leads from `CF1` to `CF2` via one or more edges in the control-flow graph. We say that `CF2` follows from `CF1` because there is a path from `CF1` to `CF2`. Note that we wish to match when there is at least one such path, so multiple paths are permissible.

We now introduce the control-flow predicate `directlyFollows` that simply matches a pair of control-flow points, `(CF1,CF2)`, for which there is single edge in the control-flow graph that will lead us from `CF1` to `CF2`.

```
directlyFollows(CF1,CF2):-
     controlFlowEdge(CF1,CF2).
```

As stated before, we wish to match any pair such that there is a path of one or more edges from the first to the second. This leads us to a simple, but incorrect, definition that is nonetheless a good example of a typical recursive query:

```
follows(CF1,CF2):-
     directlyFollows(CF1,CF2).
follows(CF1,CF2):-
     directlyFollows(CF1,MID),follows(MID,CF2).
```

The first rule-case above is what we call the base case of the recursive query. This is where `CF2` is reachable in one step, via one edge of the control-flow graph. The second rule-case is the recursive part. Here we are essentially saying that `CF2` follows from `CF1` if

there is some other control-flow point, `MID`, such that `MID` directly follows from `CF1` and that `CF2` follows from `MID`. This allows the query to follow any path that will lead us from `CF1` to `CF2`.

We can visually represent the recursive part of follows as the graph given in Figure 3.2. In this we know that there is an edge from `CF1` to `MID`. We therefore know that `CF2` follows `CF1` so long as there is some series of edges from `MID` to `CF2`, which we represent by the jagged line between these points. The dotted line down the middle of the graph is to indicate that `CF2` can be in any part of the control-flow graph so long as a direct path exists.

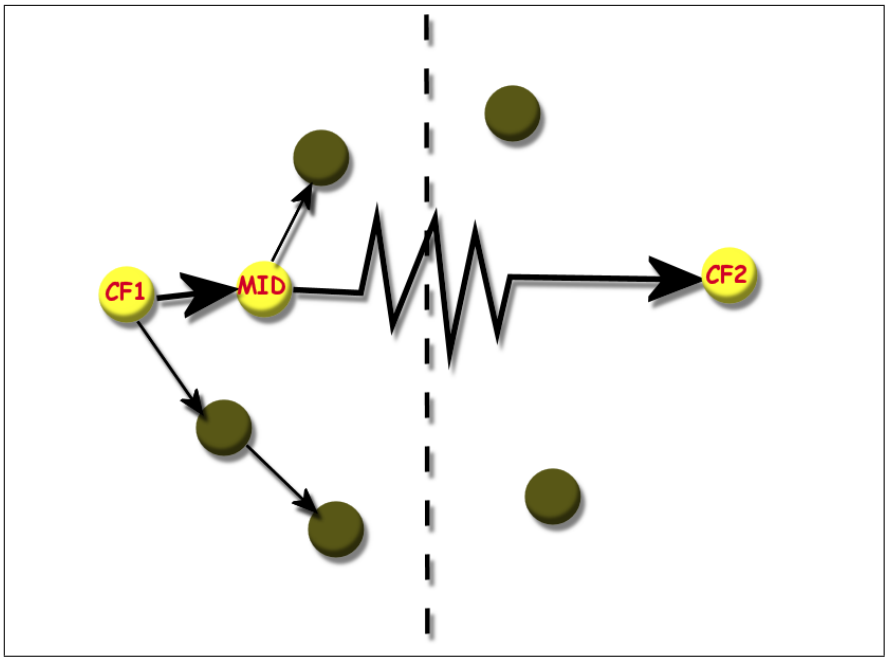


Figure 3.2: Graph diagram demonstrating recursive part of follows

The reason that this simple definition of follows is incorrect, is that it does not take into account what happens when we encounter a method-call in the control-flow graph as we attempt to find a path from one point to another. As discussed in the `methodCallToPossibilities` example previously, a method-call can result in the execution of several different implementations of the called method, depending on the exact runtime type upon which the method is invoked. Since we cannot usually statically determine the exact runtime type of a variable or expression that will evaluate to an object, we assume that the control-flow can pass to any of the methods that could theoretically be called[2].

We developed the query `methodCallToPossibilities` for just this purpose, to tell us which methods could potentially be executed as result of a method call. We thus create a simple predicate `followsOneStep` that will tell us where the control-flow can potentially lead from one point in the control-flow graph, `CF1`, by only following one edge. The edge

[2]There is some opportunity to statically determine what type(s) a variable or expression might take at runtime. This is discussed further when we look at points-to analysis.

allowed by `methodCallToPossibilities` is an implied edge, as we do not actually store this in the control-flow graph in the database.

```
followsOneStep(CF1,CF2) :-
    directlyFollows(CF1,CF2) ; methodCallToPossibilities(CF1,CF2).
```

Finally, we use `followsOneStep` to arrive at our definition for follows, as given in Listing 3.3.

```
1 follows(CF1,CF2)  :-
2     followsOneStep(CF1,CF2).
3 follows(CF1,CF2)  :-
4     followsOneStep(CF1,MID),follows(MID,CF2).
```

Listing 3.3: Datalog definition of follows.

3.3.4 Stream IOException Detection

This query is a nice and relatively straightforward example of the type of query that we can write to make use of the `follows` query. Here we examine one possible use involving the `java.io.InputStream` and `java.io.OutputStream` Java classes, and their subtypes.

`InputStream`s and `OutputStream`s in Java represent a resource which should be relinquished when it is no longer used, in order that the system can reclaim the resource. For this purpose, the method `close()` is provided in the interface to both[3]. The effect of invoking the `close()` method upon an instance of `InputStream` is that it closes the input stream and releases any system resources associated with the stream . Once this happens, any attempts to read from this stream will result in a `java.io.IOException`. Similarly, if we call `close()` on an `OutputStream` instance and then attempt to write to it, it will also cause an `IOException`.

It is desirable to have a query that could detect points in source code, where it is possible for a read/write call to be made to a stream, after it has already been closed. We proceed by writing queries to match points of interest - calls to `read()`/`write()` and `close()` - and use our previously written definition of `follows` to determine if it is possible for these to cause an `IOException`. We will proceed to write this query for `InputStream`. Writing the query for `OutputStream` will be almost identical.

We will not labour too much on the details, but we assume that we have a pre-defined query, `inputStreamSubtypeStar`, that will match all types that are subtypes of `InputStream` as well as the `InputStream` type as well. We can then match all variables (fields, parameters and local variables) that have one of these types.

```
inputStreamSubtypeStarVar(V) :-
    inputStreamSubtypeStar(T),varHasType(V,T).
```

We must now define a query that matches methods that override the method `read()`[4].

[3]The method close() is in fact specified in the java.io.Closeable interface, but both InputStream and OutputStream implement this interface and are not abstract classes.

[4]Here we discuss methods that override read(), however for a more complete and useful query we will also want to match the methods read(byte[] b) as well as read(byte[] b, int off, int len), and similarly for some of the InputStream subclasses that define other methods for reading from the stream.

```
methodOverridingInputStreamRead(M) :-
     inputStreamRead(M).
methodOverridingInputStreamRead(M) :-
     inputStreamRead(M2),overrides(M,M2).
```

In this relatively simple query we will only try to match method-calls of the form `var.m()`, in order that we can immediately associate a method invocation with a variable. It would also be possible to perform this query by associating method-calls with the heap object they might be invoked upon, by using points-to analysis (which we develop later).

As stated in Table 3.2 earlier, in the `makesMethodCall` relation we store an identifier for the expression upon which the method is invoked. In our case, we are only interested in where this expression is a variable access. This leads us to:

```
methodInvokedOnVar(MC,V) :-
     methodCallExpression(MC,E),variableAccess(E,V).
```

We now wish to match method calls to `InputStream.read()`, and subtype implementations of this method, with the variable upon which it was invoked:

```
variableCallsInputStreamRead(MC,V) :-
     methodCallTo(MC,M),methodInvokedOnVar(MC,V),
     methodOverridingInputStreamRead(M).
```

Similarly for `InputStream.close()`:

```
variableCallsInputStreamClose(MC,V) :-
     methodCallTo(MC,M),methodInvokedOnVar(MC,V),
     methodOverridingInputStreamClose(M).
```

We now combine this into a single query that will match triples of:

- A variable `V` of type `InputStream`, or any subtype thereof.
- A method-call, `MCR`, to `InputStream.read()` or any subtype definition of this method.
- A method-call, `MCC`, to `InputStream.close()` or any subtype definition of this method.

```
inputStreamSubtypeStarVarCallingReadAndClose(V,MCR,MCC) :-
     inputStreamSubtypeStarVar(V),variableCallsInputStreamRead(MCR,V),
     variableCallsInputStreamClose(MCC,V).
```

Finally, we use our definition of `follows` to match when it is possible that a method-call to `read()` occurs after a method-call to `close()` on the same variable, giving us the query in Listing 3.4. Here the use of `follows` tells us that there is a possible path through the control-flow such that the control-flow point `MCR` can come after `MCC`, meaning that an `IOException` could be thrown.

```
1 possibleIOException(V,MCC,MCR)  :-
2     inputStreamSubtypeStarVarCallingReadAndClose(V,MCR,MCC),
3     follows(MCC,MCR).
```

Listing 3.4: Datalog definition of possibleIOException.

3.3.5 Points-to Analysis

Points-to analysis is a static analysis which computes for every pointer a set of objects it may point to at runtime[32]. Java does not have "pointers" in the traditional C and C++ sense of memory cells that contain the addresses of other memory cells. Instead, the Java compiled code references memory via symbolic "handles" that are resolved to real memory addresses at run-time by the Java interpreter[15]. Java's form of pointers are commonly referred to as references, and guarantee some useful properties with regards to points-to analysis. For instance, Java references are either `null` or point to a valid instance for an object of the reference's type.

We wish to develop a query that can determine all of the possible objects that could be referenced by a particular parameter, field, local variable or expression in the source code. We follow the style taken by Whaley and Lam[40], referring to potential objects as "heap objects" and identifying them by their allocation site. This means that results from our query will be locations in source code where object creation occurs, i.e. a constructor-call. Note that we also allow the null pointer to be a heap object. We thus define our first constituent part of the query:

```
heapObj(H) :-
     constructorCall(H) ; nullLiteral(H).
```

This query will require numerous sub-queries, but most importantly we will define queries to determine what heap objects can be pointed to for expressions, for local variables and parameters, and for fields. These queries will form a mutually-recursive set of queries that will form our points-to analysis.

Before we proceed, we should note that the type of points-to analysis that we are developing here is inclusion-based and both flow-insensitive and context-insensitive. Firstly, the analysis is flow-insensitive as we do not take into account the ordering of assignment statements as they occur within a method-body. Secondly, we do not allow method executions to have different contexts according to where they were invoked. We instead assume that every method execution could have been the result of any of the locations in code where it is possibly called. As a result, the results of our query will potentially be less accurate than a flow-sensitive or context-sensitive analysis. By less accurate, we mean that we might find that a variable or expression might point to an object that it cannot. We will however, also find all correct points-to relations. The extent to which the query will be less accurate depends completely on the source-code over which the query is being ran. However, it also means that our query is simpler, cleaner, more efficient, and should scale more readily to deal with large sets of input code. The merits of context-sensitivity in a points-to analysis are discussed in more detail in the technical report by Lhoták and Hendren[25].

Our analysis is inclusion-based, as we allow different variables and expressions to point to overlapping but different locations. For example, we allow for $v1, v2 \in V$, the set of all variables, and $h1, h2 \in H$, the set of all heap objects, the three relations $(v1, h1), (v1, h2)$ and $(v2, h2)$. In a unification-based analysis this would not be allowed, as we would also require the relation $(v2, h1)$. Points-to locations form sets in a unification-based analysis, and if a variable can point to one item in the set, then necessarily it can point to them all. Figures 3.3 and 3.4 illustrate this difference. An inclusion-based analysis is therefore more accurate. Traditionally, practical points-to analyses were unification-based in order to

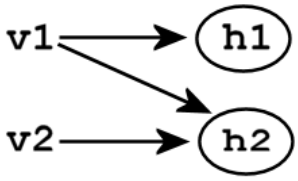


Figure 3.3: Inclusion-based

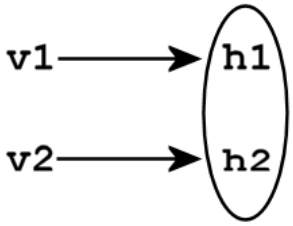


Figure 3.4: Unification-based

achieve scalability (e.g. [31, 12]), but more recent research has developed efficient inclusion-based techniques (e.g. [39]).

We first deal with points-to for expressions. We will give a definition for a query `pointsToForExpression` that will match an expression with the heap objects to which it could point. Our most simple case is when the expression itself is a heap object:

```
pointsToForExpression(E,H) :-
     heapObj(H),heapObj(E),equals(E,H).
```

Next, if the expression is a variable access, then we use our `pointsToForVar` query which we will define later. Similarly, if the expression is a field access, we use `pointsToForFieldAccess` which we will also define later.

```
pointsToForExpression(E,H) :-
     variableAccess(E,V),pointsToForVar(V,H).
pointsToForExpression(E,H) :-
     fieldAccessHolder(E,FA),pointsToForFieldAccess(FA,H).
```

Our final rule-part for `pointsToForExpression` will handle the case where we must examine further expressions. For example, if our expression is a method call, we must look at the potential return expressions that could determine the result of the method-call. Similarly, for an inline conditional expression of the form "`(exp1) ? exp2 : exp3`", the overall expression could evaluate to be the result of `exp2` or `exp3`. We first define a query `examineExpression` that will tell us for a given expression, what further expressions might determine what this expression might point to.

```
examineExpression(E,X) :-
     methodCallHolder(E,MC),methodCallToPossibilities(MC,M),
     returnExpression(_,M,X).
examineExpression(E,X) :-
     inlineConditional(E,_,X,_) ; inlineConditional(E,_,_,X).
```

We can now add a recursive rule-part for `pointsToForExpression` that will recursively call itself on the results of `examineExpression`:

```
pointsToForExpression(E,H) :-
     examineExpression(E,X),pointsToForExpression(X,H).
```

This is sufficient for an implementation of `pointsToForExpression`, assuming we have correct definitions for `pointsToForVar` and `pointsToForfield`.

The query `pointsToForVar` is used to match possible points-to relations for all non-field variables, i.e. parameters and local variables. We claim that a variable can point to anything that appears on the right-hand side of an assignment to it. That is, the heap objects that a variable may point-to is the union of heap objects that each expression that it is assigned to may in turn point to. We can therefore start by identifying places where a variable is assigned to:

```
naiveAssignmentToVarFrom(V,X) :-
     assignmentToAndFrom(_,VA,X),variableAccess(VA,V).
```

We also follow the style of Whaley et al. by modelling that method invocation arguments are in fact assignments to the relevant parameter variable[38]. For example, if a method is invoked by the call `m(x)` then we have modelled it as though the receiving method's parameter has been assigned to `x`. We add another rule-part to `naiveAssignmentToVarFrom` to model this assumed assignment. The fact that we are not just matching literal assignments is why we chose to call the query `naiveAssignmentToVarFrom`.

```
naiveAssignmentToVarFrom(V,X) :-
     params(V,_,_,N,M),methodCallToPossibilities(MC,M),
     parameterExpression(MC,X,N).
```

We now simply make use of `naiveAssignmentToVarFrom` and our earlier defined predicate `pointsToForExpression` to give our definition of `pointsToForVar`:

```
1 pointsToForVar(V,H) :-
2     naiveAssignmentToVarFrom(V,X),pointsToForExpression(X,H).
```

Listing 3.5: Datalog definition of pointsToForVar.

Points-to analysis for non-static fields is somewhat trickier than it is for local variables and parameters. The field access `a.f` is not the same as `b.f` unless `a` and `b` point to the same object. There are two straightforward options here: either we treat all field accesses as if they were to a single object's field, or we treat a non-static field as a pair consisting of a heap object and the actual field, `(H,F)`. This choice is again a trade-off between accuracy and efficiency, and various research has advocated one or other of these approaches[26, 33, 30, 32, 40]. We choose here to follow the latter method of modelling non-static fields as a pair. Therefore, instead of attempting to determine what a certain field might point to, we instead wish to determine what a certain field in a certain heap object might point to. We first define queries to identify static and non-static fields:

```
staticField(F) :-
     field(F),hasStrModifier(F,'static').
nonStaticField(F) :-
     field(F),NOT(hasStrModifier(F,'static')).
```

Similarly to our points-to analysis for a variable, we wish to examine points where a field is assigned to. However, we also wish to know what heap object it might be that is having

its field assigned to. For every heap object, `h`, that might own the field, `f`, being accessed here, we assume that there is an assignment to the pair `(h,f)`. We must therefore devise a method of determining what object might own the field being assigned to.

When a field is being accessed in the form `<expression>.f`, where the expression is of a type that declares the field `f`, we intuitively perform a points-to analysis to determine what this expression might point to. When a field is being accessed with a plain `f`, then the field is being accessed internally by the object which owns it. We can identify such cases easily as the expression field in our `fieldAccess` relation will be 0. In our analysis we treat this as being the same as `this.f`. We must therefore perform a points-to on the use of `this`. We firstly identify the method in which the field access was made — we in fact store this information in the `fieldAccess` relation for just this purpose. We then identify the method-invocation sites where this method might have been called, by using `methodCallToPossibilities`. If the matched method-call has an expression (i.e. is not 0), then the `this` might be decided by that expression.

```
possibleExpsForThisInMethod(M,X) :-
     methodCallToPossibilities(MC,M),methodCallExpression(MC,X),
     NOT(equals(X,0)).
```

However, if a matched method-call itself is an internal call and has no expression, then we must recursively match method-invocation sites where it in turn might have been called:

```
possibleExpsForThisInMethod(M,X) :-
     methodCallToPossibilities(MC,M),methodCallFrom(MC,CALLER),
     methodCallExpression(MC,0),
     possibleExpsForThisInMethod(CALLER,X).
```

To better explain the process given above, and to show where the extra accuracy is advantageous, consider the short Java program in Listing 3.6. The correct points-to result for the pair `(heapA,f)` is `heap5`, and `(heapB,f)` can point to `heap0`. We examine some of the steps that allow our query to reach this conclusion.

- We identify two assignments to the field f, on lines 5 and 17.
- Points-to analysis of the `this` on line 5 tells us that it can point to `heapA` only. To do so, we first identify where `setDefaultF()` is called. Finding one site on line 9, we realise that this is an internal method-call and we cannot determine directly. We know that the invocation of `setDefaultF()` occurs within the method `reset()`, so we find the possible sites at which `reset()` was called. This leads us to line 14 where we find `reset()` invoked upon an expression that is a variable access to `a`. A quick points-to analysis for `a` tells us that it can solely point to `heapA`. This gives us what we want. We now have that the pair `(heapA,f)` can point to `heap5`.
- Since `b` can only point to `heapB` we quickly also have that the pair `(heapB,f)` can point to `heap0`.
- We do not have that the pair `(heapA,f)` can point to `heap0`, or that `(heapB,f)` can point to `heap5`. If we considered all fields to be single entities, then both of these would (incorrectly) also have been matched.

```
public class A {
    Long f;

    public void setDefaultF() {
        this.f = new Long(5);    //Heap object heap5
    }

    public void reset() {
        setDefaultF();
    }

    public static void main(String[] args) {
        A a = new A();           //Heap object heapA
        a.reset();

        A b = new A();           //Heap object heapB
        b.f = new Long(0);       //Heap object heap0
    }
}
```

Listing 3.6: An example of internal method-calls and field accesses.

We define a predicate `pointsToForFieldAccessExpression` that can match all heap objects that might be the owner of a field for a given field access. One rule-part handles the case where there is an expression involved, and the other where there is not one, making use of `possibleExpsForThisInMethod`.

```
pointsToForFieldAccessExpression(FA,H) :-
    fieldAccess(FA,F,E,_),pointsToForExpression(E,H).
pointsToForFieldAccessExpression(FA,F,H) :-
    fieldAccess(FA,F,0,M),possibleExpsForThisInMethod(M,E),
    pointsToForExpression(E,H).
```

We can now give a concise definition for `assignmentToFieldFrom`:

```
assignmentToFieldFrom(H,F,X) :-
    assignmentToAndFrom(_,FAH,X),fieldAccessHolder(FAH,FA),
    pointsToForFieldAccessExpression(FA,H).
```

With a query for identifying assignments to a heap-object and non-static field pair, we can easily write our points-to query for a non-static field:

```
pointsToForNonStaticField(H1,F,H2)  :-
    assignmentToFieldFrom(H1,F,X),pointsToForExpression(X,H2).
```

Listing 3.7: Datalog definition of pointsToForNonStaticField.

Static fields can be treated almost identically to local variables and parameters. We know that there is only one such field, so all assignments to it are considered.

```
assignmentToStaticFieldFrom(F,X) :-
    assignmentToAndFrom(_,FAH,X),fieldAccessHolder(FAH,FA),
    fieldAccess(FA,F,_,_),staticField(F).
```

This leads us rather straightforwardly to our definition of points-to for a static field in Listing 3.8.

```
1 pointsToForStaticField(F,H) :-
2     staticField(F),assignmentToStaticFieldFrom(F,X),
3     pointsToForExpression(X,H).
```

Listing 3.8: Datalog definition of pointsToForStaticField.

We have one final part of points-to for expressions to deal with — the case where the expression is a field access. We have two types of field access, and we handle each type in a separate rule-part for our query `pointsToForFieldAccess`. Firstly, static fields can be handled simply by performing a points-to analysis on the static field:

```
pointsToForFieldAccess(FA,H) :-
    fieldAccess(FA,F,_,_),staticField(F),pointsToForStaticField(F,H).
```

Secondly: non-static field accesses. We perform a points-to analysis on the expression, and then perform a points-to analysis for each heap object and field pair found.

```
pointsToForFieldAccess(FA,H) :-
    fieldAccess(FA,F,_,_),nonStaticField(F),
    pointsToForFieldAccessExpression(FA,HP),
    pointsToForNonStaticField(HP,F,H).
```

At this stage we are almost done, but we must modify our analysis to handle arrays. Since this is a static analysis, we cannot know for certain what indices are used when various references are stored in an array or read from an array. We therefore make the assumption that any object that is stored in a particular array might be accessed when an array access is performed. For this reason we identify arrays by their creation site (e.g. `new String[6]`), and define the points-to relation for an array to be any heap object that our points-to analysis tells us might have appeared on the right-hand side of an assignment to this array. Thus, when we encounter an array access (or the array creation site), we compute the points-to analysis for the array as above. Full detail on our handling of arrays can be found in the Datalog code in Appendix A.5.

This completes all of the necessary parts for our points-to analysis.

3.3.6 Enhancing Possible Methods Called with Points-to Analysis

In our earlier efforts to implement `methodCallToPossibilities` we allowed it to match all method implementations that were stored in the system that could theoretically be executed as a result of the method-call. We can further refine this query by making use of our points-to analysis queries.

The method executed as a result of a dynamic method-call is dependent on the types upon which the method can be invoked. If we can narrow the set of possible types, then we

can improve the accuracy of our query to determine what method implementations might actually be executed by a method-call.

We first define some subsidiary queries that will be useful in the definition of this query. The query `overridesOrEqual` matches a pair `(M1,M2)` such that either `M2` overrides `M1`, or they are in fact the same method.

```
overridesOrEqual(M1,M2) :-
    overrides(M1,M2) ; (method(M1),method(M2),equals(M1,M2)).
```

We also define a query `implementsMethod` that matches when a type `T` declares a non-static method `M2` that overrides the method `M1` (or is in fact `M1`).

```
implementsMethod(T,M1,M2) :-
     declaresMethod(T,M2),overridesOrEqual(M2,M1),
     NOT(hasStrModifier(M2,'abstract')).
```

If we know that a method-call can be invoked on a type `T`, we must be able to find the method implementation that will in fact be executed by the method-call if it is invoked on a method of type `T`. If `T` implements the called method, by our definition above, then it is this method implementation that shall be executed. If it does not, then the implementation defined by the nearest supertype that implements the called-method is executed. We define the following query to find the nearest implementation `M2` of a method `M1` for a type `T`, allowing it to be the definition given by `T` itself.

```
nearestImplementation(T,M1,M2) :-
     implementsMethod(T,M1,M2).
nearestImplementation(T,M1,M2) :-
     NOT(implementsMethod(T,M1,M2)),hasSubtype(S,T),
     nearestImplementation(S,M1,M2).
```

We now write a query to determine the types upon which a particular method-call might be invoked. If the method call is invoked upon an expression, we directly perform a points-to analysis on the expression and determine the types of these.

```
possibleTypesForMethodCall(MC,T) :-
     methodCallExpression(MC,E),pointsToForExpression(E,H),
     constructorCallType(H,T).
```

When the method-call does not have an expression, such as `m()`, we treat it as if it is a call to `this.m()`. This leaves us with an identical problem as the internal field-access we dealt with while writing our points-to analysis — we must determine what the implicit `this` might point to. Similarly to our handling of fields, we find the method in which the method-invocation was made by using `methodCallFrom`. We can now reuse the `possibleExpsForThisInMethod` query we defined before, and then proceed as normal.

```
possibleTypesForMethodCall(MC,T) :-
     methodCallExpression(MC,0),methodCallFrom(MC,CALLER),
     possibleExpsForThisInMethod(M,X),pointsToForExpression(E,H),
     constructorCallType(H,T).
```

We can now give our definition of `enhancedMethodCallToPossibilities` in Listing 3.9, by finding the nearest implementation of the called method for each type upon which the method-call might have been invoked.

```
1 enhancedMethodCallToPossibilities(MC,M) :-
2     possibleTypesForMethodCall(MC,T),methodCallTo(MC,M1),
3     nearestImplementation(T,M1,M).
```

Listing 3.9: Datalog definition of enhancedMethodCallToPossibilities.

Use of this method of calculating the possible methods called for a method invocation is most likely going to be much more expensive than our original version. The accuracy gained through the use of a points-to analysis is a trade-off against the extra computation required. Applications requiring a higher degree of accuracy as a priority (e.g. analysis for optimisations) might favour the enhanced version, whereas applications that need efficiency more than absolute accuracy should stick with the original definition.

It is worth noting that we can retro-fit our points-to analysis to use our enhanced possible methods called query. Using a more accurate possible methods called query might increase the accuracy of a points-to analysis for a method-call, as we more accurately identify the return expressions that can be responsible for the result of a method-call. Also, we might increase the accuracy of a points-to analysis for a parameter, as we more accurately identify method-calls that actually result in the parameter's containing method's execution.

3.3.7 SQL Injection Detection

This query is inspired by a similar effort developed as an example query for the Program Query Language (PQL)[27, 24]. Structured Query Language (SQL) attacks form a particularly potent threat to systems that provide a web application with an underlying database store[20]. The attack, in essence, takes place when a user's input to a web application can contain SQL code and, without sufficient checks to ensure the input is safe, results in the SQL code contained in the input being executed on the web server's database. This would theoretically give a malicious user free reign over the content of the database, allowing them to view, modify or delete any information contained within. In some cases it could allow a malicious user to gain full access to a server. With many businesses holding sensitive information in databases that are connected to these web applications, this is a security threat that should be taken very seriously.

What we wish to achieve with this query is the ability to detect where possibly malicious string inputs can reach a critical point in the application, for instance the point at which they would be executed as part of an SQL query.

If we are dealing with a web server running the Java Platform, Enterprise Edition (Java EE), then our source of malicious strings is likely to be all strings returned by `HttpServletRequest.getParameter(String)`. Similarly, our critical use points might be any call to `Statement.execute(String)`. It is left as an exercise to the reader to write queries to identify the exact points at which an untrusted string might be obtained, and to identify the critical points where strings are used. We therefore assume that we have the following two queries: `criticalObtain` that matches method-calls of which the returned

string might be tainted, and `criticalUse` that matches method-calls to where a string is critically used and the expression that is passed as the critical argument of the method-call.

We now aim to write a query, `possibleInjection` that will match a triple `(MC,E,CO)` such that: `MC` is a method-call to a critical use, `E` is the critical string expression that is provided to it, and `CO` is where a critical string was obtained that might be contained in `E`. We introduce the notion of a "tainted" string as being a string that may be unsafe.

This query will consist of two parts. Firstly, we can leverage our points-to analysis from before to check if expressions/variables/fields can point to another tainted string. To do this, instead of determining method-calls that an item can point to, we instead find critical obtain method calls that an item can point to. Secondly, we add the ability to detect derivations of strings.

The objects that we will consider in our string derivation analysis are `String`, `StringBuffer` and `StringWriter`. We wish to identify all constructions of and method-calls to objects of these types (and all subtypes) that could propagate a potentially unsafe string through our system. As an example, a construction of a `StringBuffer` with a tainted `String` object as its argument will produce a tainted `StringBuffer`. Similarly, a call to `StringBuffer.append(String)` with a tainted `String` as the argument will cause the `StringBuffer` object to be tainted, but will also return a tainted `StringBuffer` (since the append method returns a reference to the same `StringBuffer`).

Due to the sheer number of different possible methods and constructors that might result in the propagation of an untrusted string, we will not give all the details here. However, the full code for this query can be found in Appendix A.7.

We require there to be two queries that identify all points of interest with regards to string derivations.

- The first, `possibleTaintedReturnFromMethodCall`, matches all method-calls that could potentially result in a tainted object being returned. It will also match the expression that could cause the returned object to be tainted. For example, for a method-call `sb.toString()` where `sb` has type `StringBuffer`, the method call will be matched with the expression `sb` as the returned object will only be tainted if `sb` is tainted. Note that this query will also cover tainted constructor calls, identifying constructions that might be tainted if a given argument is tainted (e.g. `new String(taintedString)`).

- The second query, `possibleTaintedTargetObjectAfterMethodCall`, should identify all method-calls that might result in the target of the method-call becoming tainted. For instance, if we append a tainted string to a `StringBuffer` it should be marked as tainted. This query should match two items: the expression upon which the method is called, and the expression that could cause the target to become tainted.

We shall now proceed by giving the definition of our overall query, and then continue to define the required parts. Listing 3.10 gives our definition for `possibleInjection`.

```
1 possibleInjection(MC,E,CO) :-
2     criticalUse(MC,E),taintedExpression(E,CO).
```

Listing 3.10: Datalog definition of possibleInjection.

We must therefore define a query `taintedExpression` that can determine if an expression can be tainted or not. This query will be quite similar to our definition of `pointsToForExpression` in our points-to analysis. The first possible way in which an expression can be tainted is if it is a critical obtain method call.

```
taintedExpression(E,CO) :-
     methodCallHolder(E,CO),criticalObtain(CO).
```

Next, we consider the case where the expression is a field access or a variable access. As you might expect, we simply pass this on to other queries that can determine whether the field or variable accessed can be tainted.

```
taintedExpression(E,CO) :-
     variableAccess(E,V),taintedVariable(V,CO).
taintedExpression(E,CO) :-
     fieldAccessHolder(E,FA),taintedFieldAccess(FA,CO).
```

For our last rule-part we consider the case where we must consider further expressions to determine if this expression is tainted. We already have a definition of `examineExpression` that finds further expressions to examine, so we can reuse that here. However, we also want to encompass the possibility of a tainted return from a method call, in the case that it is a string derivation method-call. We therefore define a new predicate `taintedExamineExpression` that gives two rule-parts — one that reuses `examineExpression` and the other that considers possible tainted returns.

```
taintedExamineExpression(E,X) :-
     examineExpression(E,X).
taintedExamineExpression(E,X) :-
     methodCallHolder(E,MC),possibleTaintedReturnFromMethodCall(MC,X).
```

We now give the final rule-part for `taintedExpression`, that recursively examines further expressions.

```
taintedExpression(E,CO) :-
     taintedExamineExpression(E,X),taintedExpression(X,CO).
```

This completes our rules for dealing with tainted expressions. We must now define rules that identify variables and fields that can be tainted. There are two possible ways for a variable or field to become tainted. Either the variable/field is assigned to from a tainted expression, or it is the target for a tainting method-call.

```
taintedVariable(V,CO) :-
     naiveAssignmentToVarFrom(V,X),taintedExpression(X,CO).
taintedVariable(V,CO) :-
     possibleTaintedTargetObjectAfterMethodCall(TARGET,E),
     variableAccess(TARGET,V),taintedExpression(E,CO).
```

Once more, we can treat a static field like a normal variable. This leads us very quickly to the definition below:

```
taintedStaticField(F,CO) :-
     staticField(F),assignmentToStaticFieldFrom(F,X),
     taintedExpression(X,CO).
taintedStaticField(F,CO) :-
     possibleTaintedTargetObjectAfterMethodCall(TARGET,E),
     fieldAccessHolder(TARGET,FA),fieldAccess(FA,F,_,_),
     staticField(F),taintedExpression(E,CO).
```

For a non-static field we again work with heap object and field pairs. A heap object and field pair becomes tainted when they are assigned to by a tainted expression, or when they are the target for a tainting method-call.

```
taintedNonStaticField(H,F,CO) :-
     nonStaticField(F),assignmentToFieldFrom(H,F,X),
     taintedExpression(X,CO).
taintedNonStaticField(H,F,CO) :-
     possibleTaintedTargetObjectAfterMethodCall(TARGET,E),
     fieldAccessHolder(TARGET,FA),fieldAccess(FA,F,_,M),
     possibleExpsForThisInMethod(M,H),
     nonStaticField(F),taintedExpression(E,CO).
```

This completes our SQL injection detection query.

Chapter 4

Implementation

In this section we discuss the implementation of the CodeQuest system and how it was modified during the course of this project. Note that the development of several useful queries for this system also formed a substantial part of this assignment, and these were already discussed in Section 3.3. Here we detail some of the design decisions and implementation details of interest in the extension of the CodeQuest system to extract and store the required extra information from Java source code.

4.1 Architecture

The CodeQuest system currently consists of two parts:

- The source code fact-gathering front-end of the project is responsible for extracting the required facts from the source code and storing these in a relational database. Currently, the system supports the collection of facts from Java source code. This portion is itself written in `Java`, and takes the form of a plug-in for the `Eclipse Java IDE`.
- The Datalog-to-SQL conversion and query execution mechanism. This portion of the system takes as input a set of pre-defined Datalog predicates and a specific Datalog query. The Datalog is first parsed, then some checks are performed, and then the Datalog query is converted into an SQL query for execution on the database system. Finally, the SQL query is executed and the requested results are returned. Full detail on this part of the system can be found in the original CodeQuest thesis [17]. This component is written using C#, utilising the Microsoft .NET framework.

It might seem odd that these two pieces of the system are written in different languages. The reason that the frontend is written in Java is to allow a direct comparison with the JQL query system. Writing this part in Java also allows us to leverage the functionality provided by the Eclipse plug-in model, and particularly the Eclipse Java Development Tools (JDT). The motivation for developing the backend in C# is that it is intended to integrate this system with the "Extensible Toolkit for Refactoring" that is currently being developed in the Computing Laboratory by Mathieu Verbaere [35]. Another ultimate goal of the CodeQuest

project is to integrate its full implementation as a plug-in for the Microsoft Visual Studio IDE.

In the rest of this chapter we describe the extension of the original CodeQuest fact gathering mechanism to obtain and store more detailed information about the source code being parsed, in particular information about the implied control-flow of this code.

4.2 The Eclipse Platform and Plug-in Development

The Eclipse Platform[11] is a core framework and set of services upon which plug-in extensions can be created. Plug-ins are structured bundles of code and/or data that contribute function to the system. Written in Java, development on Eclipse projects follow the open-source development model. The Eclipse Project also provides a wide variety of plug-ins for end-users to utilise. Each subsystem in the platform is itself structured as a set of plug-ins that implement some key function. The best known manifestation of these projects is in the Eclipse Software Development Kit (SDK), more commonly referred to as the Eclipse IDE. This is a combination of the Eclipse Platform and numerous Eclipse plug-ins that provide a powerful development environment for Java.

Eclipse plug-in developers are able to choose which other plug-ins are available to the system when the platform is executed. Thus, a developer can choose to start with a bare bones system — The Eclipse Rich Client Platform (RCP) — and add functionality from there, or to start with the full set of tools as provided by the Eclipse SDK. We choose to do the latter; allowing programmers to leverage the set of features provided in the standard Eclipse IDE, and add the functionality to parse working sets of Java source code, storing relevant facts into a relational database system.

The development of our CodeQuest plug-in is carried out within the Eclipse Plug-in Development Environment (PDE), a tool designed to assist developers in the creation, development, testing, debugging, and deployment of Eclipse plug-ins. From here, we make use of several hooks (extension-points of the Eclipse Platform that allow plug-in developers to add functionality) to add the necessary functionality. The Eclipse Platform Architecture can be visualised as in Figure 4.1.

4.2.1 Java Development Tooling

The Java Development Tooling (JDT) plug-in for Eclipse allows users to write, compile, test, debug, and edit programs written in the Java programming language. It provides the core functionality that allows programmers to develop Java applications on the Eclipse Platform.

A feature of the JDT plug-in is that it allows programmatic access to its parsing and compilation utilities. We may ask for a certain compilation unit (.java file) to be parsed, and receive an Abstract Syntax Tree (AST) representing the compilation unit in response. The JDT provides access to the `org.eclipse.jdt.core.dom.ASTParser` class for this purpose. We may create an ASTParser object and use it to create an AST as in Listing 4.1. Note that all classes/types mentioned in the following discussion are contained in the package `org.eclipse.jdt.core.dom` unless otherwise specified.

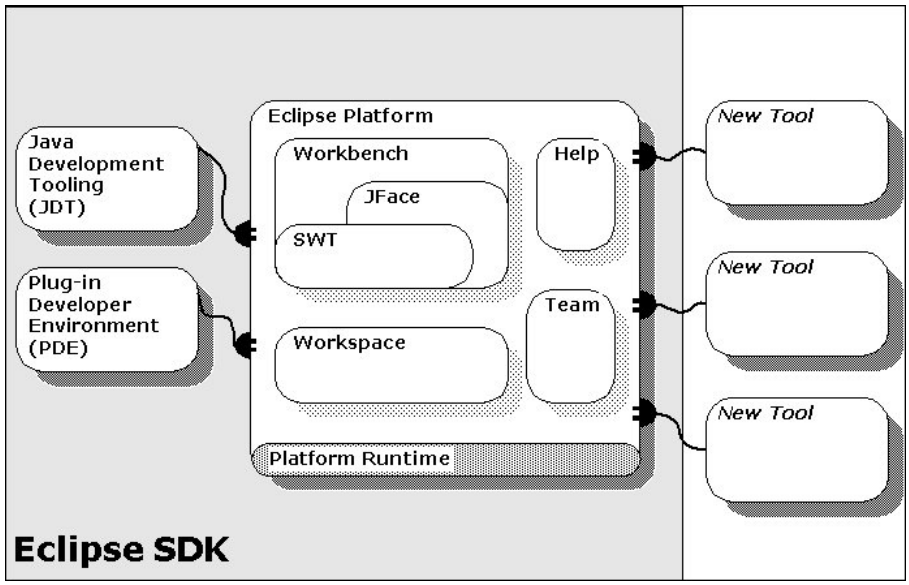


Figure 4.1: The Eclipse Platform Architecture

```
1 ASTParser parser = ASTParser.newParser(AST.JLS3);
2 parser.setSource(cu);
3 parser.setKind(ASTParser.K_COMPILATION_UNIT);
4 parser.setResolveBindings(true);
5 CompilationUnit node = (CompilationUnit)parser.createAST(monitor);
```

Listing 4.1: Java code to create an AST using the Eclipse JDT.

- Line 1 is where we construct the ASTParser object. `AST.JLS3` is a constant signifying that we wish to parse all source versions up to Java 1.5.
- On line 2 we set the source to be parsed. We are passing an object of type `ICompilationUnit`, which is an object representing an entire compilation unit.
- Nodes in an AST produced by the JDT in this manner are all subtypes of `ASTNode`. On line 3 we ask that the root node returned when we create the AST is of type `CompilationUnit`, which is the top-level ASTNode type with which we wish to begin the traversal of the AST. This action ensures that our class-cast on lines 6-7 is also safe.
- Line 4 ensures that all binding information is available in the AST generated.

4.3 The Parsing Action

All parsing of the Java source code is performed by the ASTParser class provided by the JDT plug-in. We traverse abstract syntax trees that are built for us in order to collect the

necessary facts we want from the source code. The mechanism for traversing an AST that is provided by the JDT is to subclass `ASTVisitor`. The `ASTVisitor` class is an implementation of the visitor portion of the visitor design pattern. This means it implements a method with signature `public boolean visit(XX node)`, where `XX` is an ASTNode type, for every ASTNode type that is defined in the `org.eclipse.jdt.core.dom` package. The returned boolean is used to indicate whether the child nodes of the node currently being visited should be visited also. The default implementation of all of these visit methods is to do nothing and return true, thus visiting all nodes for a given AST. For further information on the visitor design pattern, see the book where it was originally described: Design Patterns - Elements of Reusable Object-Oriented Software[13].

We define our own subclass of the ASTVisitor type, `CodeQuestASTVisitor`, and we use this to gather information as it is passed around the AST. There are in fact 83 concrete subtypes of `ASTNode` in the `org.eclipse.jdt.core.dom` package and 10 abstract classes, including `ASTNode` itself. We override the visit method for each concrete `ASTNode` type, and sometimes also change the order in which child nodes are visited by our visitor implementation. We do not give visit methods for the abstract types as, due to the mechanics of the visitor pattern and the Java Language Specification[1], a visit method for an abstract type will never be invoked if there is a visit method for all concrete subclasses — as is the case.

Figure 4.2 gives a diagram of some of the key types in the `ASTNode` class hierarchy. For full details see the Eclipse JDT API Specification, accessible via the Eclipse website.

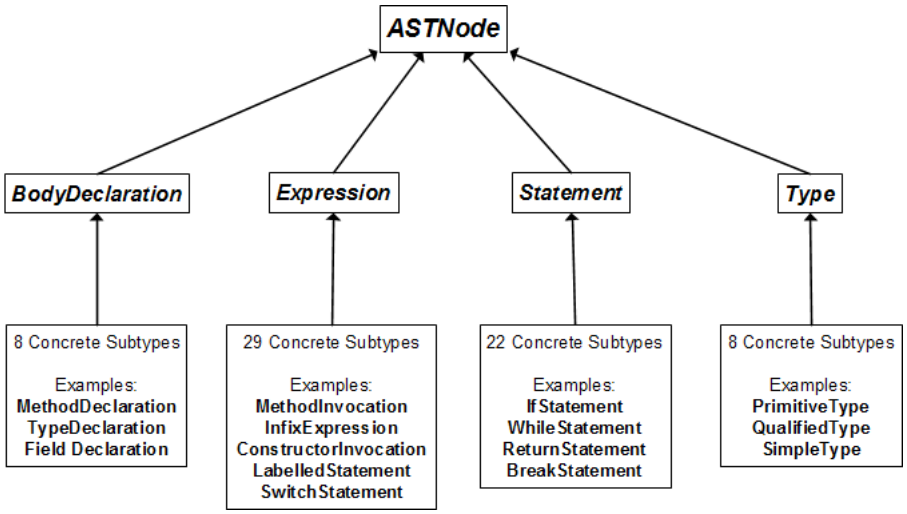


Figure 4.2: Some of the ASTNodes representing source elements

The `org.eclipse.jdt.core.dom` package and the `ASTNode` subtypes contained within correspond to a Document Object Model (DOM) of the source code of a Java program as a structured document. Figure 4.3 details what some source elements map to in an

[1]The JLS decrees that when more than one member method is both accessible and applicable to a method invocation, the most specific method is chosen. Informally, one method is more specific than another if any invocation handled by the first method could be passed on to the other one without a compile-time type error.

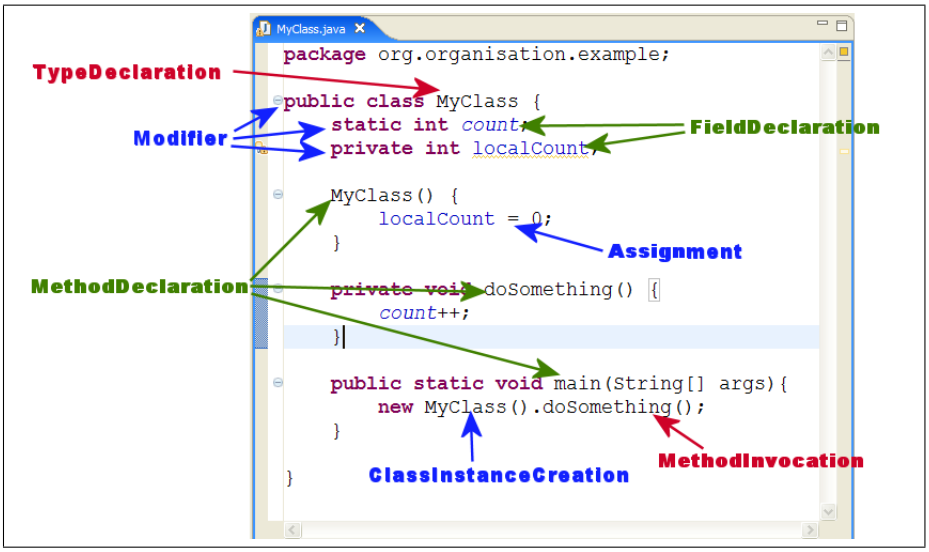


Figure 4.3: Some of the ASTNodes representing source elements

AST generated by the JDT. There would in fact be many more nodes generated by this short section of code, the diagram is just to give the reader an idea of how source code translates into `ASTNodes` for the AST formed. The structure of the document/source-code file is described by the hierarchical organisation of the AST created to represent it. Every part of the Java source code will be represented in the AST created. For example, every method-call will result in a `MethodInvocation` node and every field declared will result in a `FieldDeclaration` node. Furthermore, the location of a node in the AST is a reflection of its structural location in the source code. For instance, every `MethodDeclaration` node that is a descendant of a `TypeDeclaration` means that that method is declared in that type.

4.3.1 Unique Identifiers

It is desirable for us to be able to assign a unique identifier to every point of interest within a set of Java source code. Our points of interest will largely be nodes on the AST, or bindings for methods, types or variables.

We wish to establish a system of storing identifier mappings such that each item will correspond to a unique identifier in a one-to-one mapping. We could choose to use strings as identifiers, but this would consume an unnecessary amount of storage space and make the querying time considerably slower. Instead, we go through a two-part conversion to get a numerical unique identifier for an item.

Firstly, we derive a unique string from the item in question. For example, for a method we can use a concatenation of the signature for the type in which it is contained, the method signature and the return type. For instance:

`Ljava.lang.Object;.toString()Ljava.lang.String;`

The key properties for the mechanism by which we create a string representing an item

are: it must produce a unique string for each unique item, and that it must produce the same unique string every time for the same item — regardless of whether it is the same instantiation of the object representing the item. The second requirement prevents us from utilising `String.identityHashCode()`, as this may/will return a different value for two separate object instantiations representing the same item. We instead follow the method of concatenating sufficient information about the item so as to uniquely identify it, like in the example for `Object.toString()` given above.

We must be careful when producing unique strings for low-level code items such as if-statements, method-calls or assignments. The source-code text for many of these can be repeated in multiple locations. For instance, the method-call `a.b()` could be made in two subsequent statements in code. We must ensure that each unique item has a unique identifier, as required. We can achieve this by adding the compilation unit name and location of the item in the source file to the unique string.

Once we have a mechanism for obtaining a unique string for each item, we utilise these strings to assign a unique number to the string, and therefore to the item. We do this by keeping track of strings we have seen previously as well as their numerical mapping. When asked for the unique identifier for a certain string (i.e. for a certain item) we first check if it has been seen before, and if it has we return the previously assigned number. If it has not been seen before, we return a new unique number and also add a mapping from the string to this number.

Due to the sheer volume of numbers and strings involved, we cannot assume that we will be able to store all of these mappings in memory — we want the process to be scalable, and we also want the mappings to be non-volatile. For this reason a two-tier pooling system was established to hold mappings in memory, with a table, `fullids`, in the database backing these mappings to ensure longevity and scalability. The pooling system ensures that the most recently accessed mappings are held in memory, attempting to minimise the number of database accesses required to determine mappings for strings.

4.3.2 Storing Variable Accesses

In this section we are discussing variable accesses — places in code where we either read the reference stored in a variable or write a reference to a variable. By variable we again mean any field, parameter or local variable. Variable accesses can occur in lots of places in code, including assignments, expressions and when we pass a variable as an argument to a method-call.

When we encounter a variable access during our fact-gathering traversal we have a slight problem — the binding of the variable that is being accessed is not a unique point that can be used in our control-flow analyses. If we access the same variable in several locations, as is likely, we cannot use the same identifier for all these points as it would render our control-flow graph incorrect. Thus, we need to create an identifier that is unique for each of these points. This can be easily done using the method described in the previous section. However, we also want to know exactly what variable is being accessed at each of these points. If we use a unique identifier for each variable access as is required for our control-flow analyses, we will not be able to discern what variable is being accessed.

We get around this problem by introducing a new relation, and therefore a new table in

the database store. We create the relation `variableAccess` that stores pairs of identifiers. Each pair will consist of the identifier for a unique control-flow point that is a variable access, and the identifier for the variable that is being accessed.

4.3.3 Storing Method-call and Field Access Locations

We encounter a somewhat similar problem when we encounter method invocations and field accesses. In Java, a field access is of the form:

`[Expression.]fieldName`,

and a method invocation of the form

`[Expression.]methodName`.

With regards to the method arguments, we ignore these as they are not of interest to this discussion.

In the case of a field access, the expression can be any expression that has a type that declares a field of name `fieldName`. For method calls, the expression can be any expression that has a type that declares a method of name `methodName`. Also, for static method-calls, the expression can simply be a type name.

The problem with the above is that in terms of control-flow, the expression should be evaluated first. However, it is desirable to be able to identify the whole thing as a field access or as a method-call when we encounter it. Our solution for how to achieve both of these aims is to make the overall field access or method call a point in the control-flow graph. We create relations `methodCallHolder` and `fieldAccessHolder` that each store a pair of identifiers — the unique control-flow point's identifier and the actual field access/method-call's identifier.

4.3.4 Storing Context As We Traverse

The traversal of the AST by our ASTVisitor class is performed in a top-down fashion. This is the logical manner in which to traverse, but it leaves us with the problem of determining context when visiting nodes further down the tree. A given ASTNode stores very little in way of immediately-accessible context.

It is always possible to work out the context of a given node by an upwards traversal of the AST starting from that node. However, in many cases this is likely to be a costly endeavour, especially when it comes to determining the control-flow context for a node.

Some examples:

- When visiting an ASTNode representing a return statement we can only establish what method it is returning by repeatedly accessing parent nodes until a method declaration is reached.
- When visiting an ASTNode representing a break statement we must climb the AST until we reach a loop, switch statement or labelled statement with a matching name (if one was supplied in the break statement) in order to determine exactly where we are breaking from.
- If we encounter an ASTNode representing a number literal we must climb the AST to determine whether we are inside a boolean expression, a parameter expression or some other type of expression in order to establish where the next point in the control-flow

should be. Furthermore, if it is inside a boolean expression, for example, we must find out if the boolean expression is inside another boolean expression, or inside the guard for an if statement, loop, or inline conditional. Again, we must know all of this context in order to be able to know where the control-flow should go from this point.

We instead opt to collect context as we traverse downwards on a tree, thus hopefully saving time and reducing the complexity involved in determining context for lower-placed nodes in the AST. We developed a context storage class to provide all of the necessary mechanisms for adding context, and for retrieving the details about the current context when necessary. We create a new context for each AST we traverse as all context is specific to each compilation unit, and our root node is always an ASTNode representing a compilation unit. As we visit each node in the AST, we almost always add some context before moving down to any child nodes.

For storing the context we largely made use of:

- Variables of type `Long`.
  Long is the type we use to store a unique numerical identifier. Some examples of context stored using a Long variable: the current compilation unit, the current method, the next statement after this one, the next case in a switch statement, short circuit exit points for a boolean expression.

- Variables of type `boolean`.
  For obvious reasons we use these to store context information that can be either true or false. For example: are we inside an expression, are we inside a boolean expression, are we processing the right-hand-side of an expression.

- `Stack` variables storing `Long`s.
  These were used when the context could be nested, for example — types and loops can be nested. Examples: the current type, continue and break points for loops, labelled statements and switch statements, next statement after a block.

- `HashMap` variables mapping from `String` to `Long`.
  Two of these were used to store the break and continue points for labelled statements. Thus, when we encountered a break or continue statement that used a name, we were able to determine when the control-flow would jump to.

4.4 Control-Flow Information

The control-flow sequence for the execution of Java programs is specified by the Java Language Specification (JLS)[14]. Whilst traversing the AST for a compilation unit, we attempt to capture the control-flow sequence as dictated by the JLS. We wish to store the structure of the control-flow graph for the parsed source code, and we do so by storing the edges from the control-flow graph in the database.

The table `controlFlowEdge` is used to store these edges, containing a pair of identifiers. The first of the pair is the originating control-flow point, and the second is the destination of the control-flow after following this edge. Thus, if a pair `(CF1,CF2)` appears in the

`controlFlowEdge` table, then we are stating that it is possible for the control-flow of the program to pass from `CF1` to `CF2`.

Whilst we traverse an AST we add control-flow edges to the database starting from the current node, and use our stored context to determine where the control-flow can go next. To illustrate this action, we will look in some more detail at some examples — the control-flow within a block, the control-flow for a loop, the control-flow within a boolean expression, the control-flow for break and continue statements, and the control-flow for switch statements. There were of course other ASTNode types that required handling, but we do not include everything for the sake of brevity. These examples were picked to give a representative sample of the techniques employed and the type of considerations and design decisions made during the implementation.

Note that as a general pattern we redefine the order in which child nodes are visited for most ASTNode types. The reason for this is that it allowed us to manipulate the context in-between visiting child nodes. This is especially important with regards to the control-flow context. By controlling the sequence in which we visit each child we are able to update the context as to where the control-flow should/can go after leaving the child which we are about to visit. The general algorithm for visiting a node in this manner is given below, ignoring all details except from updating the context and visiting the child nodes.

---

initial update context
for all *child* in *children* **do**
    intermediate update context
    visit *child*
end for
final update context

Of course, the child nodes are not always in a list, in which case we explicitly visit the children according to the order in which the control-flow would visit them.

4.4.1 Blocks

A block in Java consists of a sequence of statements. Statements are terminated by a semi-colon, and the sequence can be of any length, including zero. A block is delimited at either end by braces (‘`{`’ and ‘`}`’). Blocks occur in numerous places, including: method bodies, then and else parts for if statements, and loop bodies.

The JLS tells us that control-flow passes through each of the statements in a block in order from first to last. Thus, when we reach the last control-flow point within a statement, we must add a control-flow edge from there to the start of the next statement. In order to do this, when we are processing a statement, we must always have available in context what the next statement after this one is. We can provide this context by always storing the next statement to be processed (assuming there is one after the current). We follow the general pattern in the algorithm given above, storing the next statement to be visited in context before visiting the current one.

Care must be taken when setting the context for the the last statement in the sequence. In some cases, for instance a method-body block, there is no further control-flow edges following the last statement. However, since blocks can be nested, in many cases there is

actually another statement to follow the last in the current block. We design our context store to carefully keep track of which statement follows the current block (if any). When visiting a top-level block, such as a method-body, there will be no statement following the current block. As we visit a block we push the location of the statement following it (again, if any) onto a stack. When we finish visiting a block, i.e. once we have finished visiting all child nodes of the block, we pop the next statement from the stack. In order to determine the statement that follows the last statement in a block, we look at which statement is at the top of our stack. This process is sufficient to ensure that the captured control-flow between statements is correct.

4.4.2 Loops

The three loop types present in the Java language are sufficiently similar that it requires only minor modifications to our handling of one to accommodate another. We therefore discuss while-loops in particular, and it is left as an exercise for the reader to imagine the small changes that must be made to deal with for-loops or do-loops.

Loop bodies are indeed blocks, but we must handle these slightly differently than the manner in which we handle other block types. After completing execution of a loop body, the control-flow does not pass to the following statement. Instead, the control-flow passes to the boolean expression that is the guard for the loop. Figure 4.4 demonstrates the important control-flow edges for a while-loop.

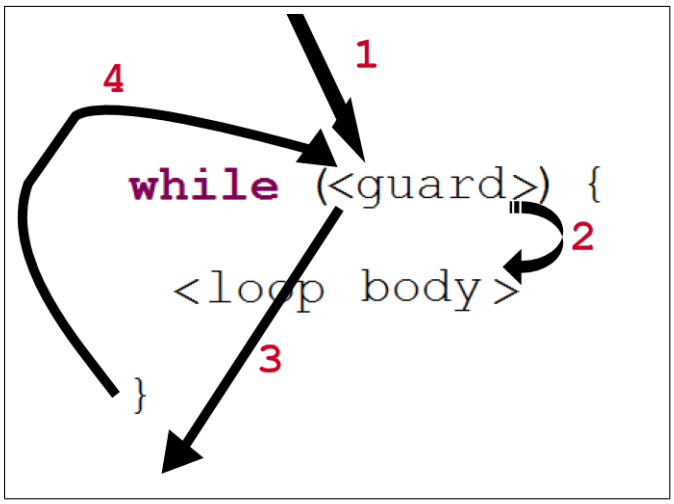


Figure 4.4: Important Control-flow Edges for a While-loop

- The control-flow edge depicted by arrow 1 in Figure 4.4 indicates the edge leading into the while-loop. The loop-guard boolean expression is evaluated first.
- Arrow 2 indicates where the control-flow goes when the guard evaluates to true.
- Arrow 3 demonstrates the control-flow exiting the loop and moving to the next statement when the guard evaluates to false.

- Finally, arrow 4 shows that when the loop body has finished executing the control-flow passes back to the loop-guard.

We must capture all of these edges to ensure that we are storing the control-flow for the loop correctly.

Before we visit the loop guard expression, we update the context to reflect the fact that we are inside a boolean expression. Part of this is to indicate where the control-flow jumps to when the expression evaluates to true, and also when it evaluates to false. This is discussed in more detail in the next section. For a while-loop, we set the point that the control-flow will move if the guard evaluates to true as the first statement in the loop body. If there are no statements in the loop body, the control-flow will return to start executing the loop guard once more. We set the point the control-flow will move to after the guard evaluates to false as the next statement after the while-loop.

Whilst visiting the loop-body block, we perform a slight hack to ensure that the control-flow returns to the loop guard after the last statement. Normally before we enter a block we update the context to store the next statement after the block (if any), as discussed before. We update the context to indicate that the next statement is in fact the loop guard. As a result, when we visit the final statement in the loop body, the control-flow will pass back to the loop guard.

4.4.3 Boolean Expressions

The control-flow involved in evaluating boolean expressions would be trivial except for the use of conditional-and (`&&`) and conditional-or (`||`) operators. Both of these are operators for use in infix expressions of type boolean. The reason that they make life more difficult in determining the control-flow edges to store is that the right-hand side of infix expressions using these operators will not always be executed. The right-hand side of an infix expression with a conditional-and operator will only be evaluated if the left-hand side evaluates to true. Conversely, the right-hand side of an infix expression with a conditional-or operator will only be evaluated if the left-hand side evaluates to false. Essentially, evaluation of a boolean expression will stop as soon as the result can be determined. This behaviour is commonly referred to as short-circuit evaluation.

Figure 4.5 demonstrates the effect of short-circuit evaluation on the control-flow for a boolean expression. In this, the boolean expression in question is the expression for an if statement. Arrow number 3 represents the "short-circuit", if `exp1` evaluates to false, the control-flow will immediately skip to the else part. If we were dealing with a conditional-or (`||`) instead of the conditional-and, then arrow 3 would point to the then part, as a short-circuit would result in the expression evaluating to true.

Before entering a boolean expression, we store in the context what the next control-flow points are according to whether the expression evaluates to true or false. We also store in the context that we are visiting the left-hand side of the expression before doing so, and similarly for the right-hand side. This allows us to determine if a short-circuit of the boolean evaluation can occur when we are visiting the descendant nodes of the boolean expression, and if so, where to create edges to.

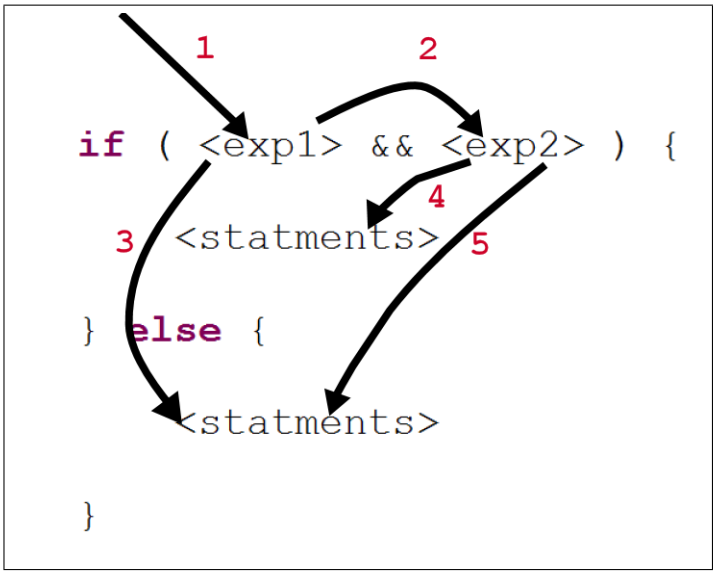


Figure 4.5: Important Control-flow Edges for Boolean Expression in an If-Statement

4.4.4 Break and Continue Statements

According to the JLS "a break statement with no label attempts to transfer control to the innermost enclosing switch, while, do, or for statement of the immediately enclosing method or initializer block; this statement, which is called the break target, then immediately completes normally." When a break statement provides a label, the break target is the enclosing labelled statement with the same name.

In terms of storing context and computing control-flow we handle the cases where a label is provided with the break statement, and the cases where one is not, completely separately. We first examine the case where no label is provided. The fact that we are looking for the innermost switch, while, do, or for statement suggests that we should use a stack structure to store the necessary context. We do just that, by adding the control-flow point to the stack for every such statement that we enter. This control-flow point is the point to which the control-flow would jump after encountering a break statement inside this statement, but not inside any other statement. This is normally the statement immediately following the switch, while, do, or for statement. Thus, when we encounter a break statement with no label, we simply ask the context for the break-to point that is stored at the top of the stack.

For break statements that include the label of the statement to break from, a simple stack structure is insufficient. For this reason we also maintain a map in the context that maps a string to the relevant break-to point. Thus, when we enter a labelled statement we also add the mapping from the statement's label to its break-to point. When we exit a labelled statement, we remove this mapping. This mechanism allows us to find the break-to point easily by getting the stored mapping for the label provided with a break statement.

Continue statements are largely similar to break statements, but can only occur within loops — while, do, or for statements. The continue target is defined almost identically to the break target. What's different though is that a statement of this form transfers the

control-flow back to the loop guard (in the case of while and do statements) or to the first updater expression (in the case of a for statement). Similarly to how we handled storing the necessary context for break-to points, we use both a stack and a map for the continue-to points. When we enter a loop we add the continue-to point to the stack, and when we leave the loop we pop the continue-to point off the stack. We also add a mapping to the continue-to point from the labelled statement's name when we enter one, and remove the mapping when we exit one. Once again, this allows us to determine from our context where the control-flow should go when we encounter a continue statement.

4.4.5 Switch Statements

The JLS decrees that "a switch statement transfers control to one of several statements depending on the value of an expression." For obvious reasons, the control-flow first passes to this expression to compute its value. From here, the computed value is compared against (possibly) multiple constant values — those provided by the case statements. The computed value is compared against each of the constant values in turn, until there is a match, in which case control is passed to that case statement.

Since we do not know which case statement might match, we assume that each case statement might or might not be a match. We provide a control-flow edge from the switch statement's expression to the first case statement. We must also provide a control-flow edge from each case statement to the next case statement, as the control-flow will pass to the next case if a case is not a match.

Once a match is made with a particular case, the control-flow moves to execute the statements that follow this case. We note that all statements following a matched case are executed, even those following subsequent cases, until the end or a break statement is reached. For example, in Listing 4.2, if the value of `x` is 1 then both "`one`" and "`two`" will be printed to the standard out, i.e. "`onetwo`".

```
1 switch (x) {
2     case 1:
3         System.out.print("one");
4     case 2:
5         System.out.print("two");
6         break;
7     case 3:
8         System.out.print("three");
9 }
```

Listing 4.2: An Example Switch Statement.

Ensuring all of these properties are adhered to is made a somewhat more difficult task by the fact that all statements of the switch statement, both case and non-case, are stored in a single list. To alleviate this we provide some helper methods to: find the next case in the list, and find the next non-case in the list. These simply iterate over the list from a given starting point to find the next case or non-case statement. Before visiting each statement in the list, we store in the context the next case statement and the next non-case statement

(if there is more of either). We also add to the context the fact that we are within a switch statement before processing any of the statements contained within.

Now if we encounter a case statement we: insert a control-flow edge from it to the next case-statement and to the next non-case statement (assuming there are more of either of these). Both of these points are stored in the context. However, if a case we encounter is the default case, we do not insert a control-flow from it to the next case, but just to the next non-case. Also, if we encounter a case after the default case we do not insert any control-flow edges to it, as it will never be reached. This covers the cases where the case statement matches and where it doesn't.

Finally, if we encounter a non-case statement that we know is directly inside a switch statement we insert a control-flow edge from that statement to the next non-case statement for the switch, as given from our context.

Chapter 5

Benchmarks

In this section we examine the performance of some of the queries we developed in section 3.3. We first examine the performance of our simple IOException detection query which combines a fairly straightforward matching process with some control-flow analysis. We then look at the performance of our points-to query with regards to that of `bddbddb`, which has shown to have an efficient and scalable implementation. We also compare the relative running times of our original and enhanced possible methods called query, by looking at how they perform when identifying direct method calls.

5.1 Test Environment

- All tests in the following sections were performed on an Intel Pentium M 1.7Ghz laptop with 1.5Gb of RAM, running Microsoft Windows XP SP2.
- The Java virtual machine used was the Java HotSpot(TM) Client VM (build 1.5.0_06-b05, mixed mode).
- The CodeQuest Eclipse plug-in was tested using Eclipse Platform 3.1.2.
- The version of the Joeq tool used was the latest taken from its CVS repository, as of 20th August 2006.
- The version of `bddbddb` used was the JAR file version taken from its website on 22nd August 2006.

5.2 Test Inputs

As input for our tests we used a variety of open-source Java programs. The programs varied in size and complexity, and are summarised in Table 5.1.

Program name	Version	Description	LOC
Regexp	1.4	Regular Expression Library	3264
FreeCS	1.2	Chat Server	15986
J2SSH	0.2.7	SSH Library	39666
iText	1.4.3	PDF Manipulation Library	88529

Table 5.1: Programs Used

5.3 Results

5.3.1 Stream IOException Detection

For our first test we examined the performance of our IOException detecting query. We expected the performance of this query to scale fairly linearly with the size of input code, although it would possibly vary erratically depending on how much control-flow analysis it must perform. We ran the query on our four test programs, with the running times given in Figure 5.1.

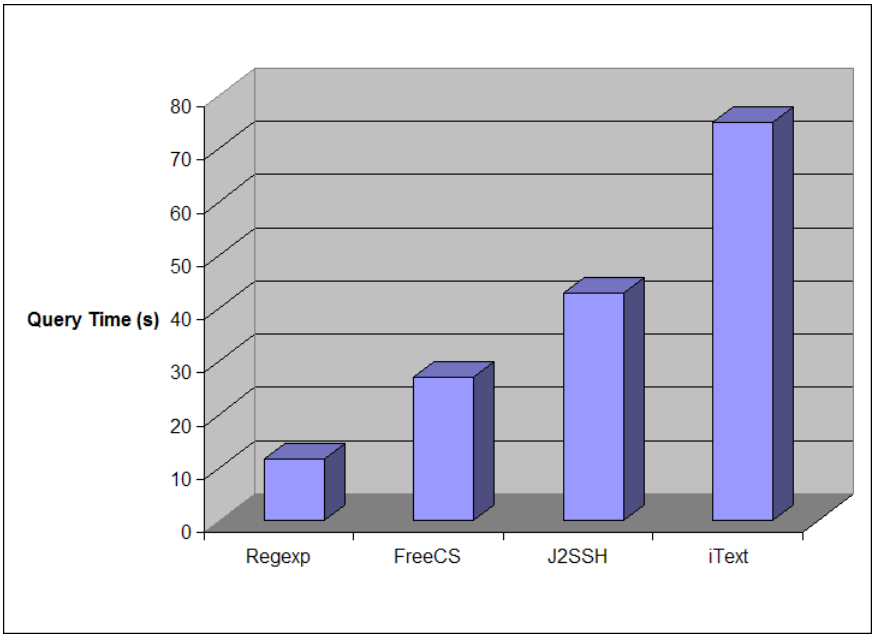


Figure 5.1: Stream IOException Detection Running Times

These results indicate that the time taken to execute the query is directly correlated to the size of the input code. This is as we expected, as in some ways we are just matching points of interest to us. However, once we match a pair of calls to `read`/`write` and `close`, we then examine the control-flow graph, attempting to find a path in a certain direction between the two points. If we must examine a large section of the control-flow graph to

determine whether such a path exists, it could result in the query time being considerably longer.

5.3.2 Points-to Analysis

Before we examine any of the performance results of running a points-to analysis on CodeQuest, it is important that we take stock of some of its limitations. CodeQuest operates at the source-code level and collects facts about each and every `.java` file given as input, but will not examine anything outside of this input. Thus, even if an input class references an object or method that is defined outside of the input, we do not collect facts about the referenced object/method. As a result, the use of objects in libraries and jars appear as somewhat of a "black box". For this reason, we include external method-calls as valid points for our points-to. If our points-to analysis determines that a variable or expression might point to an external method-call, we can say no more than the fact that it points to whatever is returned by that method.

It is also worth noting that that the information stored in our CodeQuest system is quite different to that stored by the Joeq tool for use in analyses with `bddbddb`. For instance, Joeq creates the relation that identifies all methods that could potentially be executed for a certain method-call (i.e. the call graph). The CodeQuest analysis calculates this relation as part of the query execution.

The Joeq tool `bddbddb` operates on byte-code, requiring all of the program's classes to be on the classpath. It then takes a list of classes, for which to generate the relations for. We make two remarks on this: Firstly, it means that Joeq is able to analyse all classes utilised by the program, including library classes, and include all of these in the relations produced. Secondly, it unfortunately means that all of this information must be loaded and processed at once. This can potentially cause an `OutOfMemoryError`, if a sufficiently large program is supplied. Joeq had problems processing one of the test programs, even with a heap size of 512Mb. When collecting the relations for `iText` neither 512Mb nor 768Mb was sufficient, but completed when given 1024Mb. Thus, this technique works fine for small to mid-sized programs, but given very large programs it will run into memory difficulties.

Due to the reasons discussed above, the comparison of the running times of the CodeQuest points-to query against the `bddbddb` version is not an exact comparison. Rather, it is intended to give an indication of how our query performs on CodeQuest, even if it is doing less work. After all, CodeQuest is performing a similar query, but not examining every class that `bddbddb` is.

Note that for all executions of `bddbddb` a heap size of 512Mb was used.

The times we give for `bddbddb` are from when the program starts to when it terminates. That means that it includes loading the initial relations and any other preliminary processing, converting the Datalog to internal BDD operations, "solving" — when it repeatedly applies inference rules until a fixed point is reached, and also writing the relations to disk afterwards. Similarly for CodeQuest the times given include loading the Datalog and converting it to an SQL query, executing the query, and also writing the results to disk.

Figure 5.2 shows the running times for the points-to analyses. We can immediately see that the `bddbddb`'s analysis appears to scale much more readily to deal with larger bodies of code. We will make a couple of remarks about this:

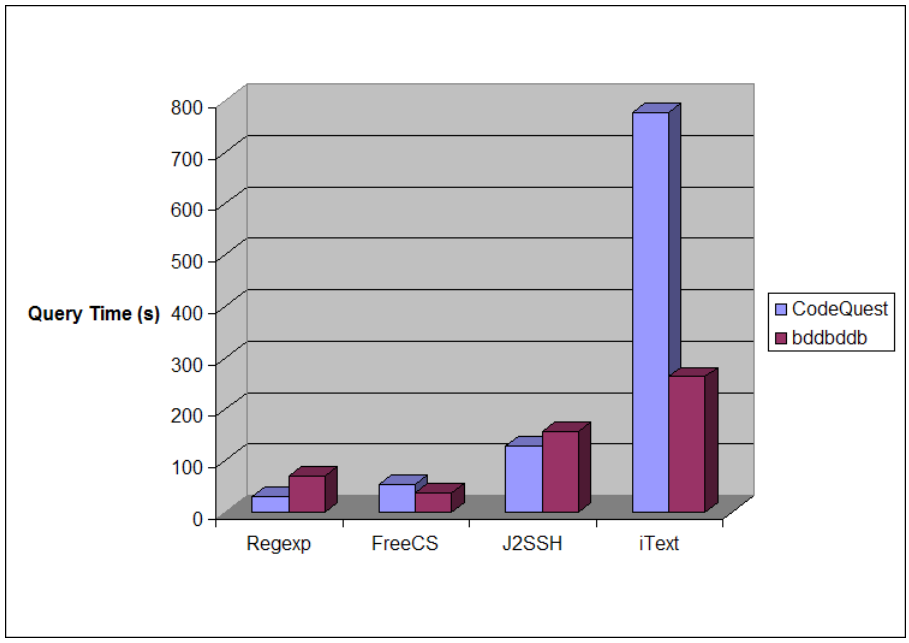


Figure 5.2: Points-to Analysis Running Times

Firstly, `bddbddb` is a specialised tool, designed with these types of analyses in mind. BDDs were originally designed for hardware verification purposes, allowing the storage of a large number of states that share many similarities in a compact and efficient manner[5]. It has been shown that BDDs are also a suitable data structure for use in a points-to analysis[4], due to this compactness and efficiency. Furthermore, previous empirical experiments with running points-to analyses with the `bddbddb` tool have allowed the choice of an effective ordering of variables. An effective ordering will reduce the size of the relations as well as the execution time required for BDD operations[24] — i.e. is an optimisation in both time and space required.

The points-to analysis performed on CodeQuest is largely unoptimised. Our Datalog definitions make heavy use of mutual recursion, and we currently apply no optimisations to this type of recursion[1]. There are optimisations that can be applied to recursion, such as the simplified version of the `magic sets` algorithm, `Closure Fusion`[2]. We suspect that adding this or other optimisations to the use of mutual recursion should greatly decrease the time required to execute this analysis, and make it more competitive with `bddbddb`.

5.3.3 Possible Methods Called

We now compare our two separate implementations for determining the possible methods called by a given method-call. We developed these queries in sections 3.3.1 and 3.3.6. For

[1]The ability to use mutual recursion is a very recently added feature to CodeQuest, optimisation of its use should follow shortly.

[2]The implementation of Closure Fusion for regular recursion is detailed in the original CodeQuest thesis[17].

each, we calculate the (method-call, method) pairs for every method-call present in the given program. Remembering that our enhanced version performs a points-to analysis, we expect this to take longer to execute.

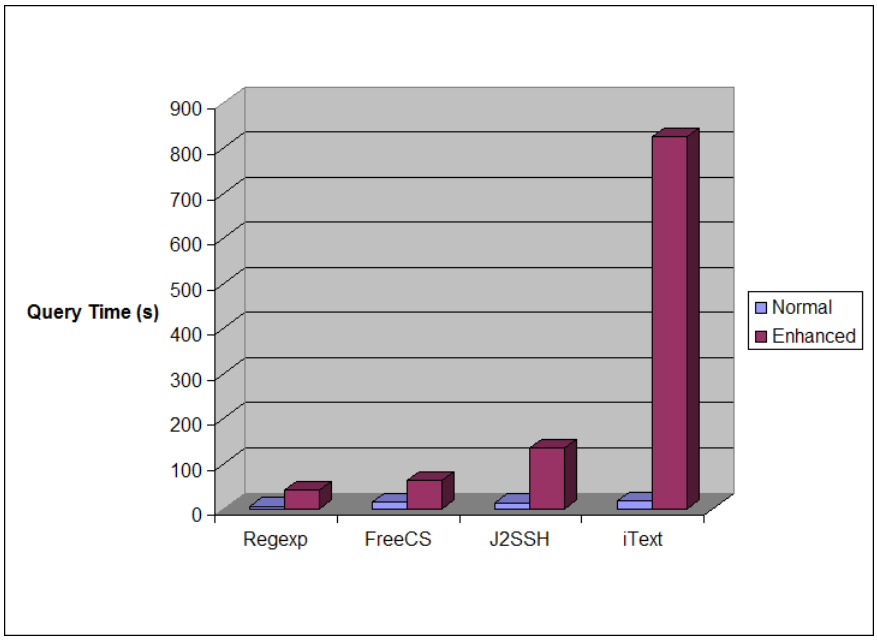


Figure 5.3: Possible Methods Called Running Times

Figure 5.3 shows us that, as expected, our normal technique performs significantly faster. Furthermore, it appears that our non-enhanced version scales remarkably well to handle large bodies of code, which is obviously highly desirable.

Our enhanced version scales very similarly to the points-to analysis, suggesting that an execution of this query will normally take just slightly longer than a points-to analysis of the same code. This is largely positive, as it means that any optimisations that affect our points-to analysis will have a knock-on affect to this enhanced version of our query. Thus, we can expect that optimisation on the use of mutual recursion will have a significant affect here also.

It would appear that currently the improved accuracy that is achieved by the enhanced possible methods called query is significantly outweighed by the additional time required to execute it, especially when the input program is large. Thus, only when the extra accuracy is very important is the enhanced version likely to be worthwhile.

Chapter 6

Conclusion

6.1 Summary Of Work Completed

This thesis has detailed the extension of the CodeQuest querying system to collect and utilise a finer-grained level of detail from source code than was used before. We have discussed the design of the system and some of the implementation details and decisions made during the development process, with particular note to the generation of the control-flow graph for Java source code.

We have shown several useful queries that can be used to take advantage of the new facts that are collected and stored. We have detailed the development of these queries, showing how queries can be developed in a modular and logical fashion. These queries are quite complex, including a points-to analysis, yet can be defined in a relatively short amount of Datalog code.

The results presented in the previous chapter suggest that the performance of the CodeQuest system on these types of complex query is reasonable, even if not state-of-the-art. Our points-to analysis may well suffer from poor performance when supplied with very large programs, however, its running time is quite acceptable for inputs up to a reasonable size. Furthermore, future optimisations on the use of mutual recursion should significantly improve the efficiency of this analysis, and its scalability.

In conclusion, we have further demonstrated the strengths of a flexible querying system by extending CodeQuest and allowing a whole new set of queries to be handled by it.

6.2 Future Work

The CodeQuest system is developing into a mature and highly useful querying tool. Work is ongoing by the original author as part of his DPhil research, and much progress has been made, even during the duration of this project. We now suggest some possible future work, improvements and extensions, that are worth considering as some of the next steps in the development of this tool.

6.2.1 Further Queries

We have presented several interesting queries in this thesis that can be practically run on the CodeQuest system. There are, however, many more possibilities for useful queries that can be performed — the literature is full of examples of queries that could be attempted. The more functional queries that are available immediately with the tool, the more the tool will be immediately useful to users.

6.2.2 Optimisations Specific To These Queries

This project has further shown that CodeQuest can represent and execute complex queries on large sets of input code. However, the length of time taken to execute some of the more complex queries certainly leaves room for improvement. It is worth spending some time examining CodeQuest's query evaluation mechanism with regards to these queries, looking for techniques that could optimise the execution time required. Specifically, the application of optimisations for use of mutual recursion

6.2.3 BDD fact storage

Work on the `bddbddb` tool has shown that it is highly efficient and suitable for the execution of certain types of analyses. Since `bddbddb` also uses Datalog for specifying queries, it would be a very useful asset if the CodeQuest fact-gathering mechanism could also output these facts in a BDD format. The CodeQuest Eclipse plug-in is designed in a manner so as to allow further fact-storage types to be added easily. This would allow the direct comparison of database versus BDD implementations for evaluating a wide-range of queries, as the input to both would be almost identical. Furthermore, the fact that `bddbddb` is implemented entirely in Java means that it could be easily integrated into the CodeQuest Eclpise plug-in.

6.2.4 User Interface

Having a useful user interface for a querying tool is essential for improving its usability. If users are to run regular queries, they do not want to have to exit their programming environment and run command-line programs each time.

CodeQuest's fact collecting system is currently integrated with the Eclipse IDE, and users can choose to parse selected bodies of code with ease. Unfortunately, CodeQuest does not currently have an integrated system for executing queries and displaying their results. Ideally, the results of a CodeQuest query would be displayed in a manner similar to JQuery, using built-in Eclipse APIs for displaying program elements in a hierarchical tree. Users would be able to select a result, and have the IDE open the containing source file and jump directly to the item in question. It is for this reason that we store the source locations of most items in the database.

As CodeQuest's Datalog-to-SQL conversion and querying mechanism is implemented separately in C#, it is not as simple as invoking a method and examining the results obtained. Results could however be passed back to the plug-in in a number of ways, the simplest being using the standard output to communicate, and a simple text structure or xml to convey the results. From here the plug-in could parse the results and set up the relevant results pane.

Other user interface enhancements could fall under the category of helping the programmer in constructing a query. Firstly, a method for viewing all the pre-defined predicates in the system would be useful, allowing a programmer to browse and search for predicates as well as read a description/notes on each. A well-designed query-input dialog would also help to make the query writing process simpler. Parsing the input Datalog and underlining errors will help greatly with debugging. Determining the variables used in a query and, in a similar manner to how JQuery does, allowing the user to select which variables are of interest and an order in which to include them in the results.

A query writer could also be aided by some automatic Datalog generation. Writing a predicate/query to identify a specific point in the source code often takes the use of several predicates. If we wish to write a query that identifies several points, this can make the query complicated quite quickly. For example, if we wanted to write a query to identify a specific method, we might pedantically write a query as below:

```
?q(M) :-
     package(P),hasName('org.query.test'),hasChild(P,C),
     class(C),hasName(C,'TheClass'),hasChild(C,M),
     method(M),hasName(M,'execute').
```

Of course, identifying the package most likely won't be needed, but we will still require the use of five predicates to identify the method in the class. A feature could be added to the Eclipse plug-in to abstract some of this detail away from the user. For instance, a facility whereby the user selects the location in code and via a context menu is able to request that the location be identified in their query. The program should then prompt for the name of the variable they wish to be bound to this location. For even further abstraction, and to reduce the complexity of the query-text, the tool could instead write a separate predicate to match the code location. The user could then select an identifiable name for this predicate (e.g. "`executeMethod`") and this would be inserted into their query, with the definition included in an auxiliary file that would be implicitly included when the query is executed.

An alternative approach to avoiding the need to write such long queries to identify simple points in the code, would be to investigate the use of alternative language to Datalog for specifying points like this, if a user desired. The use of a representation that could be easily converted to Datalog, could allow users to specify these types of points more easily and more concisely, but still allow their use in the CodeQuest system. For example, it has been shown that AspectJ pointcuts can be converted to Datalog suitable for use with CodeQuest[1], meaning that they are a suitable candidate.

6.2.5 Further Fact Collection

We briefly mentioned in our Benchmark chapter that the relations created by the `Joeq` tool for use in `bddbddb` analyses were not identical to those collected by CodeQuest. For example, `Joeq` explicitly stores all methods that could be potentially executed by a method-call whereas for CodeQuest we calculate this information on the fly at query-time. The reason that CodeQuest does not currently store such a relation, is due to the fact that it cannot be computed immediately whilst we are traversing the AST. The details of all types and methods must be available before such a relation could be calculated.

It would be possible for the CodeQuest fact collection mechanism to run a second round of fact-collecting once all ASTs have been processed. This would involve querying the facts stored in the database and creating new relations accordingly from the results.

It is possible that such a technique might be more time-consuming than it is worth, but it is certainly worth investigating the relative merits with regard to how storing these extra details at the time the facts are collected would affect both query-execution time and the time taken to parse the input code. If there was a justification for this extra step, but not in all circumstances, it could be given as an option to the user.

6.2.6 Concurrent Development Features

In a concurrent development environment, where several programmers are working on the one body of code, each developer tends to have a version of the program on their own machine as well as access to a shared repository (CVS or similar) from where they submit or check-out code. A code repository system could be extended to include elements of CodeQuest, as well as a database, so that it will store the usual CodeQuest facts about the code stored within. If this was done in such a way as to include versioning information, it would facilitate the ability of programmers to execute queries on different versions of the same code. Programmers could run queries on old versions and also compare results between versions. This would be especially useful with regards to software metrics.

Bibliography

[1] Pavel Avgustinov, Elnar Hajiev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Matheiu Verbaere. Semantics of Static Pointcuts in AspectJ. Available from `http://progtools.comlab.ox.ac.uk/`, July 2006.

[2] *BCEL*. `http://jakarta.apache.org/bcel/`.

[3] *bddbddb*. `http://bddbddb.sourceforge.net/`.

[4] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. *SIGPLAN Not.*, 38(5):103–114, 2003.

[5] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[6] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The c information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, 1990.

[7] *CodeQuest*. `http://progtools.comlab.ox.ac.uk/projects/codequest`.

[8] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the Conference on Domain-Specific Languages*, pages 229–242. USENIX Assoc., October 1997.

[9] Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems — a case study. *SIGPLAN Not.*, 31(5):117–126, 1996.

[10] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.

[11] *Eclipse*. `http://www.eclipse.org/`.

[12] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 253–263, New York, NY, USA, 2000. ACM Press.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, June 2005.

[15] James Gosling and Henry McGilton. The java language environment. White paper, Sun Microsystems, May 1996.

[16] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. Fast, flexible syntactic pattern matching and processing. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 144, Washington, DC, USA, 1996. IEEE Computer Society.

[17] Elnar Haijev. Codequest - source code querying with datalog. Master's thesis, Oxford University Computing Laboratory, September 2005. Available at `http://progtools.comlab.ox.ac.uk/projects/codequest/`.

[18] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[19] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris de Volder. *CodeQuest*: Querying source code with datalog. In *Object-Oriented Programming Languages and Systems (OOPSLA) Companion*. ACM Press, 2005.

[20] William G. J. Halfond and Alessandro Orso. Combining static analysis and runtime monitoring to counter sql-injection attacks. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[21] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM Press.

[22] *joeq*. `http://joeq.sourceforge.net/`.

[23] *JQuery*. `http://www.cs.ubc.ca/labs/spl/projects/jquery/`.

[24] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM Press.

[25] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? Technical report, Sable Research Group, School of Computer Science, McGill University, October 2005.

[26] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for java. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 73–79, New York, NY, USA, 2001. ACM Press.

[27] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM Press.

[28] Edward McCormick and Kris De Volder. Jquery: finding your way through tangled code. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 9–10, New York, NY, USA, 2004. ACM Press.

[29] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *Software Engineering*, 20(6):463–475, 1994.

[30] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java based on annotated constraints. Technical Report DCS-TR-424, Rutgers University, Nov 2000.

[31] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[32] Mirko Streckenbach and Gregor Snelting. Points-to for java: A general framework and an empirical comparison. Technical report, University Passau, November 2000.

[33] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

[34] *TyRuBa*. `http://tyruba.sourceforge.net/`.

[35] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: a scripting language for refactoring. In Dieter Rombach and Mary Lou Soffa, editors, *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.

[36] Kris De Volder. Jquery: A generic code browser with a declarative configuration language. In *PADL*, pages 88–102, 2006.

[37] John Whaley. Joeq: a virtual machine and compiler infrastructure. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 58–66, New York, NY, USA, 2003. ACM Press.

[38] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog and binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780. Springer-Verlag, November 2005.

[39] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, September 2002.

[40] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, June 2004.

Appendix A

Datalog Queries

A.1 Possible Methods Called

```
methodCallToPossibilities(MC,M) :-
     methodCallTo(MC,M),NOT(hasStrModifier(M,‘abstract’)).
methodCallToPossibilities(MC,M) :-
     methodCallTo(MC,M1),overrides(M,M1),
     methodCallExpressionType(MC,T),hasSubtypePlus(T,S),
     declaresMethod(S,M),NOT(hasStrModifier(M,‘abstract’)).
```

A.2 Detecting Direct Method-calls

```
possiblyCallsTwoOrMore(MC) :-
     methodCallToPossibilities(MC,M1),
     methodCallToPossibilities(MC,M2),NOT(equals(M1,M2)).

directMethodCallTo(MC,M) :-
     methodCallToPossibilities(MC,M),
     NOT(possiblyCallsTwoOrMore(MC)).
```

A.3 Follows

```
    followsOneStep(CF1,CF2) :-
         directlyFollows(CF1,CF2) ; methodCallToPossibilities(CF1,CF2).

    follows(CF1,CF2)  :-
         followsOneStep(CF1,CF2).
    follows(CF1,CF2)  :-
         followsOneStep(CF1,MID),follows(MID,CF2).
```

A.4 Stream IOException Detection

```
assignmentToVar(V,CF) :-
     assignmentToAndFrom(CF,VA,_),variableAccess(VA,V).
assignmentToVar(F,CF) :-
     assignmentToAndFrom(CF,FAH,_),fieldAccessHolder(FAH,FA),
     fieldAccess(FA,F,_,_).

followsNoAssignmentToVar(V,CF1,CF2) :-
     followsOneStep(CF1,CF2),var(V).
followsNoAssignmentToVar(V,CF1,CF2) :-
     followsOneStep(CF1,MID),followsNoAssignmentToVar(V,MID,CF2),
     NOT(assignmentToVar(V,MID)).

possibleIOException(V,MCC,MCR) :-
     inputStreamSubtypeStarVarCallingReadAndClose(V,MCR,MCC),
     followsNoAssignmentToVar(V,MCC,MCR).
possibleIOException(V,MCC,MCW) :-
     outputStreamSubtypeStarVarCallingWriteAndClose(V,MCW,MCC),
     followsNoAssignmentToVar(V,MCC,MCW).
```

A.5 Points-to Analysis

```
heapObj(H) :-
     constructorCall(H) ; nullLiteral(H) ; stringLiteral(H) ; arrayCreation(H,_).

examineExpression(E,X) :-
     methodCallHolder(E,MC),methodCallTo(MC,MORIG),nonStaticMethod(MORIG),
     methodCallToPossibilities(MC,M),returnExpression(_,M,X).
examineExpression(E,X) :-
     methodCallHolder(E,MC),methodCallTo(MC,M),staticMethod(M),
     returnExpression(_,M,X).
examineExpression(E,X) :-
     inlineConditional(E,_,X,_) ; inlineConditional(E,_,_,X).
examineExpression(E,X) :-
     castExpression(E,X).
examineExpression(E,X) :-
     thisExpression(E,M),possibleExpsForThisInMethod(M,X).
examineExpression(E,X) :-
     arrayCreation(E,_),pointsToForArray(E,X).


pointsToForExpression(E,H) :-
     heapObj(H),heapObj(E),equals(E,H).
pointsToForExpression(E,H) :-
     variableAccess(E,V),pointsToForVar(V,H).
pointsToForExpression(E,H) :-
     arrayAccess(E,_),pointsToForArrayAccess(E,ARR),pointsToForArray(ARR,H).
```

```
pointsToForExpression(E,H) :-
     examineExpression(E,X),pointsToForExpression(X,H).
pointsToForExpression(E,H) :-
     fieldAccessHolder(E,FA),pointsToForFieldAccess(FA,H).

pointsToForFieldAccess(FA,H) :-
     fieldAccess(FA,F,_,_),staticField(F),pointsToForStaticField(F,H).
pointsToForFieldAccess(FA,H) :-
     fieldAccess(FA,F,EXP,_),nonStaticField(F),pointsToForExpression(EXP,HP),
     pointsToForNonStaticField(HP,F,H),heapObj(H).
pointsToForFieldAccess(FA,H) :-
     fieldAccess(FA,F,E,M),nonStaticField(F),equals(E,0),
     possibleExpsForThisInMethod(M,EXP),pointsToForExpression(EXP,HP),
     pointsToForNonStaticField(HP,F,H),heapObj(H).

possibleExpsForThisInMethod(M,X) :-
     methodCallToPossibilities(MC,M),makesMethodCall(MC,_,_,X,_,_,_,_),
     NOT(equals(X,0)).
possibleExpsForThisInMethod(M,X) :-
     methodCallToPossibilities(MC,M),makesMethodCall(MC,CALLER,_,0,_,_,_,_),
     possibleExpsForThisInMethod(CALLER,X).

naiveAssignmentToVarFrom(V,X) :-
     assignmentToAndFrom(_,VA,X),variableAccess(VA,V).
naiveAssignmentToVarFrom(V,X) :-
     params(V,_,_,N,M),methodCallToPossibilities(MC,M),parameterExpression(MC,X,N).
naiveAssignmentToVarFrom(V,X) :-
     params(V,_,_,N,C),constructor(C),makesThisCall(TC,_,C,_,_,_),
     parameterExpression(TC,X,N).
naiveAssignmentToVarFrom(V,X) :-
     params(V,_,_,N,C),constructor(C),makesSuperCall(SC,_,C,_,_,_),
     parameterExpression(SC,X,N).
naiveAssignmentToVarFrom(V,X) :-
     params(V,_,_,N,C),constructor(C),makesConstructorCall(CC,_,C,_,_,_,_),
     parameterExpression(CC,X,N).

assignmentToFieldFrom(H,F,X) :-
     assignmentToAndFrom(_,FAH,X),fieldAccessHolder(FAH,FA),
     fieldAccess(FA,F,E,_),pointsToForExpression(E,H).
assignmentToFieldFrom(H,F,X) :-
     assignmentToAndFrom(_,FAH,X),fieldAccessHolder(FAH,FA),fieldAccess(FA,F,0,M),
     possibleExpsForThisInMethod(M,E),pointsToForExpression(E,H).

assignmentToStaticFieldFrom(F,X) :-
     assignmentToAndFrom(_,FAH,X),fieldAccessHolder(FAH,FA),
     fieldAccess(FA,F,_,_),staticField(F).

staticField(F) :-
     field(F),hasStrModifier(F,'static').
```

```
nonStaticField(F) :-
     field(F),NOT(hasStrModifier(F,'static')).

pointsToForNonStaticField(H1,F,H2) :-
     heapObj(H1),nonStaticField(F),assignmentToFieldFrom(H1,F,X),
     pointsToForExpression(X,H2).

pointsToForStaticField(F,H) :-
     staticField(F),heapObj(H),assignmentToStaticFieldFrom(F,X),
     pointsToForExpression(X,H).

pointsToForVar(V,H) :-
     naiveAssignmentToVarFrom(V,X),pointsToForExpression(X,H).

pointsToForArrayAccess(ARA,H) :-
     arrayAccess(ARA,AR),arrayAccess(AR,_),
     pointsToForArrayAccess(AR,H).
pointsToForArrayAccess(ARA,H) :-
     arrayAccess(ARA,AR),NOT(arrayAccess(AR,_)),
     pointsToForExpression(AR,H),arrayCreation(H,_).
pointsToForArrayAccess(ARA,H) :-
     arrayAccess(ARA,AR),NOT(arrayAccess(AR,_)),pointsToForExpression(AR,ARRAY),
     arrayCreation(ARRAY,_),pointsToForArray(ARRAY,H).

pointsToForArray(ARR,H) :-
     arrayCreation(ARR,_),assignmentToAndFrom(_,LHS,RHS),
     pointsToForExpression(LHS,ARR),pointsToForExpression(RHS,H).
pointsToForArray(ARR,H) :-
     arrayCreation(ARR,_),arrayInits(ARR,E),pointsToForExpression(E,H).
```

A.6 Enhancing Possible Methods Called

```
overridesOrEqual(M1,M2) :-
    overrides(M1,M2) ; (method(M1),method(M2),equals(M1,M2)).

implementsMethod(T,M1,M2) :-
     declaresMethod(T,M2),overridesOrEqual(M2,M1),
     NOT(hasStrModifier(M2,'abstract')).

nearestImplementation(T,M1,M2) :-
     implementsMethod(T,M1,M2).
nearestImplementation(T,M1,M2) :-
     NOT(implementsMethod(T,M1,M2)),hasSubtype(S,T),
     nearestImplementation(S,M1,M2).

possibleTypesForMethodCall(MC,T) :-
     methodCallExpression(MC,E),pointsToForExpression(E,H),
     constructorCallType(H,T).
```

```
possibleTypesForMethodCall(MC,T) :-
     methodCallExpression(MC,0),methodCallFrom(MC,CALLER),
     possibleExpsForThisInMethod(CALLER,X),pointsToForExpression(X,H),
     constructorCallType(H,T).

enhancedMethodCallToPossibilities(MC,M) :-
     possibleTypesForMethodCall(MC,T),methodCallTo(MC,M1),
     nearestImplementation(T,M1,M).
```

A.7 SQL Injection Detection

```
charSequenceType(T) :-
     interface(T,'CharSequence').
stringTypesAndSubclasses(T) :-
     charSequenceType(S),hasSubtypeStar(S,T).

methodOrConstructorHasParameterNumberAndType(M,N,T) :-
     params(_,_,T,N,M).

possibleTaintedConstructor(C) :-
     stringTypesAndSubclasses(T),declaresConstructor(T,C),
     methodOrConstructorHasParameterNumberAndType(C,0,PT),
     stringTypesAndSubclasses(PT).

//If E is tainted, then so is the object created by the constructor call CC
possibleTaintedConstructorCall(CC,E) :-
     constructorCallTo(CC,C),possibleTaintedConstructor(C),
     parameterExpression(CC,E,0).

tmcriotMethodNames(M) :-
     hasName(M,'toString') ; hasName(M,'toLowerCase') ;
     hasName(M,'toUpperCase') ; hasName(M,'substring') ;
     hasName(M,'intern') ; hasName(M,'append') ;
     hasName(M,'appendCodePoint') ; hasName(M,'delete') ;
     hasName(M,'deleteCharAt') ; hasName(M,'insert') ;
     hasName(M,'replace') ; hasName(M,'subSequence').

tmcriotMethods(M) :-
     method(M),tmcriotMethodNames(M),stringTypesAndSubclasses(T),
     declaresMethod(T,M).

tmcriaOtMethodNames(M) :-
     hasName(M,'append') ; hasName(M,'concat').
tmcria1tMethodNames(M) :-
     hasName(M,'replaceFirst') ; hasName(M,'replace') ;
     hasName(M,'replaceLast') ; hasName(M,'replaceAll') ;
     hasName(M,'insert').
```

```
tmcria2tMethodNames(M) :-
     hasName(M,'replace').

tmcria0tMethods(M) :-
     stringTypesAndSubclasses(T),declaresMethod(T,M),method(M),tmcria0tMethodNames(M),
     methodOrConstructorHasParameterNumberAndType(M,0,PT),stringTypesAndSubclasses(PT).
tmcria1tMethods(M) :-
     stringTypesAndSubclasses(T),declaresMethod(T,M),method(M),tmcria1tMethodNames(M),
     methodOrConstructorHasParameterNumberAndType(M,1,PT),stringTypesAndSubclasses(PT).
tmcria2tMethods(M) :-
     stringTypesAndSubclasses(T),declaresMethod(T,M),method(M),tmcria2tMethodNames(M),
     methodOrConstructorHasParameterNumberAndType(M,2,PT),stringTypesAndSubclasses(PT).

//If E is tainted, then so is the object returned by this method.
possibleTaintedReturnFromMethodCall(MC,E) :-
     methodCallTo(MC,M),tmcria0tMethods(M),parameterExpression(MC,E,0).
possibleTaintedReturnFromMethodCall(MC,E) :-
     methodCallTo(MC,M),tmcria1tMethods(M),parameterExpression(MC,E,1).
possibleTaintedReturnFromMethodCall(MC,E) :-
     methodCallTo(MC,M),tmcria2tMethods(M),parameterExpression(MC,E,2).
possibleTaintedReturnFromMethodCall(MC,E) :-
     makesMethodCall(MC,_,M,E,_,_,_,_),tmcriotMethods(M).
possibleTaintedReturnFromMethodCall(MC,E) :-
     methodCallToPossibilities(MC,M),returnExpression(_,M,E).

tomcia0tMethodNames(M) :-hasName(M,'append').
tomcia1tMethodNames(M) :-hasName(M,'insert').
tomcia2tMethodNames(M) :-hasName(M,'replace').

tomcia0tMethods(M) :-
     stringTypesAndSubclasses(T),declaresMethod(T,M),method(M),tomcia0tMethodNames(M),
     methodOrConstructorHasParameterNumberAndType(M,0,PT),stringTypesAndSubclasses(PT).
tomcia1tMethods(M) :-
     stringTypesAndSubclasses(T),declaresMethod(T,M),method(M),tomcia0tMethodNames(M),
     methodOrConstructorHasParameterNumberAndType(M,1,PT),stringTypesAndSubclasses(PT).
tomcia2tMethods(M) :-
     stringTypesAndSubclasses(T),declaresMethod(T,M),method(M),tomcia0tMethodNames(M),
     methodOrConstructorHasParameterNumberAndType(M,2,PT),stringTypesAndSubclasses(PT).

possibleTaintedTargetObjectAfterMethodCall(TARGET,E) :-
     makesMethodCall(MC,_,M,TARGET,_,_,_,_),methodCallTo(MC,M),tomcia0tMethods(M),
     parameterExpression(MC,E,0).
possibleTaintedTargetObjectAfterMethodCall(TARGET,E) :-
     makesMethodCall(MC,_,M,TARGET,_,_,_,_),methodCallTo(MC,M),tomcia1tMethods(M),
     parameterExpression(MC,E,1).
possibleTaintedTargetObjectAfterMethodCall(TARGET,E) :-
     makesMethodCall(MC,_,M,TARGET,_,_,_,_),methodCallTo(MC,M),tomcia2tMethods(M),
     parameterExpression(MC,E,2).
```

```
//Example criticalObtain and criticalUse for J2EE
//Will probably want to identify further methods that could be critical
//For example Statement.executeUpdate etc, and not just execute
criticalObtain(MC) :-
     interface(I),hasName(I,'ServletRequest'),hasChild(I,M),
     method(M),hasName(M,'getParameter'),methodCallToPossibilities(MC,M).

criticalUse(MC,E) :-
     interface(I),hasName(I,'Statement'),hasChild(I,M),
     method(M),hasName(M,'execute'),methodCallToPossibilities(MC,M),
     parameterExpression(MC,E,0).

taintedExpression(E,CO) :-
     methodCallHolder(E,CO),criticalObtain(CO).
taintedExpression(E,CO) :-
     variableAccess(E,V),taintedVariable(V,CO).
taintedExpression(E,CO) :-
     fieldAccessHolder(E,FA),taintedFieldAccess(FA,CO).
taintedExpression(E,CO) :-
     taintedExamineExpression(E,X),taintedExpression(X,CO).

taintedExamineExpression(E,X) :-
     examineExpression(E,X).
taintedExamineExpression(E,X) :-
     methodCallHolder(E,MC),possibleTaintedReturnFromMethodCall(MC,X).

taintedFieldAccess(FA,H) :-
     fieldAccess(FA,F,_,_),staticField(F),taintedStaticField(F,H),heapObj(H).
taintedFieldAccess(FA,H) :-
     fieldAccess(FA,F,EXP,_),nonStaticField(F),pointsToForExpression(EXP,HP),
     taintedNonStaticField(HP,F,H).
taintedFieldAccess(FA,H) :-
     fieldAccess(FA,F,0,M),nonStaticField(F),possibleExpsForThisInMethod(M,EXP),
     pointsToForExpression(EXP,HP),taintedNonStaticField(HP,F,H).

taintedVariable(V,CO) :-
     naiveAssignmentToVarFrom(V,X),taintedExpression(X,CO).
taintedVariable(V,CO) :-
     possibleTaintedTargetObjectAfterMethodCall(TARGET,E),
     variableAccess(TARGET,V),taintedExpression(E,CO).

taintedStaticField(F,CO) :-
     staticField(F),assignmentToStaticFieldFrom(F,X),
     taintedExpression(X,CO).
taintedStaticField(F,CO) :-
     possibleTaintedTargetObjectAfterMethodCall(TARGET,E),
     fieldAccessHolder(TARGET,FA),fieldAccess(FA,F,_,_),
     staticField(F),taintedExpression(E,CO).
```

```
taintedNonStaticField(H,F,CO) :-
     nonStaticField(F),assignmentToFieldFrom(H,F,X),
     taintedExpression(X,CO).
taintedNonStaticField(H,F,CO) :-
     possibleTaintedTargetObjectAfterMethodCall(TARGET,E),
     fieldAccessHolder(TARGET,FA),fieldAccess(FA,F,_,M),
     possibleExpsForThisInMethod(M,H),
     nonStaticField(F),taintedExpression(E,CO).

//And finally!
possibleInjection(MC,E,CO) :-
criticalUse(MC,E),taintedExpression(E,CO).
```