Dynamic Data Flow Analysis for Object Oriented Programs

Andrew Cain
Swinburne University of Technology
Faculty of Information and Communication Technologies
P.O. Box 218, Hawthorn, Victoria 3122, Australia
acain@swin.edu.au
Abstract

There are many tools and techniques to help developers debug and test their programs. Dynamic data flow analysis is such a technique. Existing approaches for performing dynamic data flow analysis for object oriented programs have tended to be data focused and procedural in nature. An approach to dynamic data flow analysis that used object oriented principals would provide a more natural solution to analysing object oriented programs.

Dynamic data flow analysis approaches consist of two primary aspects; a model of the data flow information, and a method for collecting action information from a running program. The model for data flow analysis presented in this thesis uses a meta-level object oriented approach. To illustrate the application of this meta-level model, a model for the Java programming language is presented. This provides an instantiation of the meta-level model provided. Finally, several methods are presented for collecting action information from Java programs.

The meta-level model contains elements to represent both data items and scoping components (i.e. methods, blocks, objects, and classes). At runtime the model is used to create a representation of the executing program that is used to perform dynamic data flow analysis. The structure of the model is created in such a way that locating the appropriate meta-level entity follows the scoping rules of the language. In this way actions that are reported to the meta-model are routed through the model to their corresponding meta-level elements.

The Java model presented contains classes that can be used to create the runtime representation of the program under analysis. Events from the program under analysis are then used to update the model. Using this information developers are able to locate where data items are incorrectly used within their programs.

Methods for collecting action information from Java programs include source code instrumentation, as used in earlier approaches, and approaches that use Java byte code transformation, and the facilities of the Java Platform Debugger Architecture. While these approaches aimed to achieve a comprehensive analysis, there are several issues that could not be resolved using the approaches covered. Of the approaches presented byte code transformation is the most practical.
Acknowledgements

This work was carried out under the supervision of Professor Tsong Yueh Chen and Professor Douglas Grant. I would like to thank them for their guidance and thoughtful comments throughout this project.

I would like to extend many thanks to Dr. Jean-Guy Schneider for his support, advice, and for the many hours of discussions we have had on this and other topics. I would also like to thank him for proof reading this thesis, and for describing swamps and fish.

Lastly I would like to thank my wife, extended family, and pets for putting up with my long absences, both physical and mental, while completing this thesis and research. Thank you for your support and understanding.
Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the thesis. To the best of my knowledge this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Signed ............................................................

Dated ............................................................
Contents

I  Introducing Data Flow Analysis  1

1  Introduction  2
   1.1  Motivation  .................................................. 2
   1.2  Contribution and Limitations  ................................ 3
   1.3  Outline  ...................................................... 4

2  Principles of Data Flow Analysis  5
   2.1  Data Flow in Testing  ........................................ 5
   2.2  Data Flow Analysis  .......................................... 6
   2.3  Dynamic Data Flow Analysis  ................................. 6
       2.3.1  Structured Dynamic Data Flow Analysis ..................... 9
       2.3.2  Object Oriented Dynamic Data Flow Analysis ............... 10
   2.4  Requirements for Data Flow Analysis  ......................... 10
       2.4.1  Approaches for Targeted Analysis  .......................... 12
   2.5  New Approaches  ............................................. 15

II  Modeling  16

3  Existing Modeling Approaches  17
   3.1  Dynamic Data Flow Analysis for C++  ......................... 17
       3.1.1  Overview  ............................................. 17
       3.1.2  The Tables  ........................................... 18
       3.1.3  Anomaly Detection  .................................... 20
       3.1.4  Evaluation  ............................................ 22
   3.2  Data Flow Analysis For Java  ................................. 24
### 3.2.1 Identifying Variables ............................................. 24
### 3.2.2 Analysis ........................................................... 27
### 3.2.3 Instrumentation ................................................... 29
### 3.2.4 Evaluation .......................................................... 29
### 3.3 Summary .............................................................. 30

### 4 Object Oriented Data Flow Analysis ........................................ 31
#### 4.1 Meta Model Approach ............................................... 31
#### 4.2 Implementing the Technique ......................................... 33
##### 4.2.1 Variable Analysis ............................................... 34
##### 4.2.2 Modeling Scoping Structures ................................... 35
##### 4.2.3 Searching the Model ............................................. 36
##### 4.2.4 Program to Model Interaction ................................... 37
#### 4.3 Implementation Requirements ........................................ 38
##### 4.3.1 Meta-Model Requirements ....................................... 38
##### 4.3.2 Communication Requirements .................................... 39

### 5 Java Model Implementation ............................................. 40
#### 5.1 Variable Analysis for Java .......................................... 40
#### 5.2 Scoping Structures for Java .......................................... 41
##### 5.2.1 Objects ............................................................ 42
##### 5.2.2 Classes and Interfaces ........................................... 44
##### 5.2.3 Arrays ............................................................. 45
##### 5.2.4 Methods and Blocks .............................................. 45
##### 5.2.5 Static Initializers .................................................. 48
##### 5.2.6 Java Model ........................................................ 48
#### 5.3 Scoping Component Containers ....................................... 49
##### 5.3.1 Heap Space ........................................................ 50
##### 5.3.2 Method Stack ...................................................... 50
#### 5.4 Searching in the Java Model .......................................... 51
##### 5.4.1 Processing Actions ............................................... 51
##### 5.4.2 Local Variables .................................................... 52
##### 5.4.3 Fields ............................................................... 52
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.4</td>
<td>Arrays</td>
<td>57</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary</td>
<td>58</td>
</tr>
<tr>
<td>III</td>
<td>Collecting Action Information</td>
<td>59</td>
</tr>
<tr>
<td>6</td>
<td>Program to Model Interactions</td>
<td>60</td>
</tr>
<tr>
<td>6.1</td>
<td>Program Instrumentation</td>
<td>60</td>
</tr>
<tr>
<td>6.2</td>
<td>Instrumentation Issues</td>
<td>61</td>
</tr>
<tr>
<td>6.3</td>
<td>Existing Approaches</td>
<td>62</td>
</tr>
<tr>
<td>6.3.1</td>
<td>C++ Probes</td>
<td>62</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Java Probes</td>
<td>64</td>
</tr>
<tr>
<td>6.4</td>
<td>Summary</td>
<td>75</td>
</tr>
<tr>
<td>7</td>
<td>Program to Java Model Implementation</td>
<td>76</td>
</tr>
<tr>
<td>7.1</td>
<td>Interface to the Model</td>
<td>77</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Managing Methods</td>
<td>78</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Instrumenting Variable Access</td>
<td>82</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Handling Blocks</td>
<td>87</td>
</tr>
<tr>
<td>7.2</td>
<td>Source Code Instrumentation</td>
<td>88</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Identifying Objects</td>
<td>88</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Classes and Objects</td>
<td>93</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Targeted Analysis</td>
<td>94</td>
</tr>
<tr>
<td>7.3</td>
<td>Byte Code Instrumentation</td>
<td>95</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Identifying Objects and Probe Methods</td>
<td>96</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Example Byte Code Instrumentation</td>
<td>97</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Targeted Analysis</td>
<td>97</td>
</tr>
<tr>
<td>7.4</td>
<td>Java Platform Debugger</td>
<td>98</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Identifying Objects</td>
<td>98</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Issues with JPDA</td>
<td>98</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Targeted Analysis</td>
<td>99</td>
</tr>
<tr>
<td>7.5</td>
<td>Summary and Recommendations</td>
<td>99</td>
</tr>
</tbody>
</table>
IV Conclusions and Future Work 101

8 Conclusions 102

8.1 Model for Data Flow Analysis ................................. 103
8.2 Data Flow Analysis for Java ................................. 104
8.3 Comprehensive Analysis ................................. 106

9 Future Work 107

V Appendices 109

A Source Code Examples 110

A.1 C++ Source Code Sample ................................. 111
A.2 Java Source Code Sample ................................. 112

B Java Model Classes 113

B.1 DFA_Variable ........................................... 114
B.2 DFA_Object ............................................ 115
B.3 DFA_Class ............................................ 116
B.4 DFA_Array ........................................... 117
B.5 DFA_Method ......................................... 118
B.6 DFA_Block ........................................... 119
B.7 DFA_StaticInitializer ................................. 120
B.8 HeapSpace ........................................ 121
B.9 MethodStack ........................................ 122
List of Figures

2.1 State chart representing the state machine used for data flow analysis .......... 8
2.2 Targeted Analysis Aspects Illustration .............................................. 13

3.1 State chart for performing dynamic data flow analysis for C++ ............... 21
3.2 Extended state chart ................................................................. 28

4.1 Meta Meta Model ................................................................. 32

5.1 Variable state class ............................................................... 41
5.2 Object Scoping Component ..................................................... 43
5.3 Class and Object Scoping Components ....................................... 44
5.4 Scoping Component for Arrays .................................................. 45
5.5 Method and Block Scoping Components ....................................... 47
5.6 Static Initializer Scoping Component ......................................... 48
5.7 Data flow analysis classes ....................................................... 49
5.8 Heap and Stack ........................................................................ 50
5.9 Heap Space class ...................................................................... 50
5.10 Method Stack class ................................................................. 51

6.1 determineObject Sequence Diagram ............................................... 73

7.1 The DFA class .......................................................................... 78

8.1 Meta Meta Model defined in Chapter 4 ............................................. 104
8.2 Java data flow analysis classes from Chapter 5 ..................................... 105

B.1 DFA_Vatiable Class ................................................................. 114
B.2 DFA_Object class ................................................................. 115
List of Tables

2.1 Method void foo(Type y) { ... y = z; ... } is called using foo(x), where Type can be replaced with any value or reference type. ................................. 12
2.2 Parameter classification table. .......................................................... 13
3.1 Variable register table ....................................................................... 19
3.2 Class access table. ........................................................................... 19
3.3 Partial action table. .......................................................................... 20
3.4 Action table for operator <<. ............................................................. 22
4.1 Events and Messages ......................................................................... 34
5.1 Events and Messages for Java ............................................................. 40
7.1 Transforming local variables into fields ........................................... 99
B.1 Methods of DFA_Variable ................................................................. 114
B.2 Methods of DFA_Object .................................................................... 115
B.3 Methods of DFA_Class ...................................................................... 116
B.4 Methods of DFA_Array ...................................................................... 117
B.5 Methods of DFA_Method ................................................................. 118
B.6 Methods of DFA_Block ..................................................................... 119
B.7 Methods of DFA_StaticInitializer ................................................... 120
B.8 Methods of HeapSpace .................................................................... 121
B.9 Methods of MethodStack ................................................................. 122
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Data flow analysis example.</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Instrumented data flow analysis example.</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Accessing private data in C++</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Friend functions in C++.</td>
<td>23</td>
</tr>
<tr>
<td>3.3 Implicitly Defined illustration</td>
<td>28</td>
</tr>
<tr>
<td>4.1 Pointer State Identifiers.</td>
<td>35</td>
</tr>
<tr>
<td>4.2 Probe Example</td>
<td>37</td>
</tr>
<tr>
<td>5.1 Array Sample Code</td>
<td>46</td>
</tr>
<tr>
<td>5.2 Implementation of <code>processAction</code> in DFA_Block</td>
<td>52</td>
</tr>
<tr>
<td>5.3 Implementation of <code>processAction</code> in DFA_Method</td>
<td>53</td>
</tr>
<tr>
<td>5.4 Code to locate a field in Java</td>
<td>54</td>
</tr>
<tr>
<td>5.5 Implementation for <code>processAction</code> in DFA_Object</td>
<td>55</td>
</tr>
<tr>
<td>5.6 Alternative implementation of DFA_Object’s <code>processAction</code></td>
<td>56</td>
</tr>
<tr>
<td>5.7 Accessing another object’s fields</td>
<td>57</td>
</tr>
<tr>
<td>6.1 Source Code Instrumentation and Exceptions</td>
<td>61</td>
</tr>
<tr>
<td>6.2 Source code probes in C++</td>
<td>62</td>
</tr>
<tr>
<td>6.3 Main from C++ Point source code.</td>
<td>64</td>
</tr>
<tr>
<td>6.4 Probes at program startup</td>
<td>65</td>
</tr>
<tr>
<td>6.5 Probes for the start of methods</td>
<td>66</td>
</tr>
<tr>
<td>6.6 Probes within methods</td>
<td>66</td>
</tr>
<tr>
<td>6.7 Probes at the end of methods</td>
<td>67</td>
</tr>
<tr>
<td>6.8 Object construction probes</td>
<td>68</td>
</tr>
<tr>
<td>6.9 Probes referencing return values</td>
<td>68</td>
</tr>
<tr>
<td>6.10 Object destruction probes</td>
<td>69</td>
</tr>
</tbody>
</table>
6.11 Probes at program termination ........................................ 70
6.12 Probes for reference variables ........................................ 70
6.13 Identifying the current object ....................................... 71
6.14 Instrumentation with exceptions ................................. 74
6.15 Incorrect reporting of object destruction ..................... 75
7.1 Method Start Instrumentation .................................. 80
7.2 Class Method Started ............................................. 81
7.3 Object Method Started ........................................... 81
7.4 Constructor Started ............................................... 82
7.5 Instrumenting for Variables ..................................... 83
7.6 Define and Reference .............................................. 84
7.7 Object Define and Reference ................................... 85
7.8 Array Define and Reference .................................... 85
7.9 Super Define and Reference ..................................... 86
7.10 Blocks and Exceptions ........................................... 87
7.11 Object Identification via hashCode ............................ 91
7.12 Object Identification with weak references .................. 92
7.13 Instrumented code for class and object creation .......... 93
7.14 Instrumented code for initialized fields ....................... 93
7.15 Instrumented Code of Listing 6.1 ............................... 96
7.16 Java byte code of Listing 7.15 .................................. 97
7.17 Java byte code instrumented version of Listing 6.1 ....... 97
A.1 C++ Point source code. ........................................ 111
A.2 Java Point source code. .......................................... 112
Part I

Introducing Data Flow Analysis
Chapter 1

Introduction

This thesis presents a new approach for managing and collecting information for data flow analysis of object oriented programs. This chapter introduces the motivation behind this research and outlines the contents of the remaining chapters.

1.1 Motivation

Program testing and debugging are complex tasks within the development life cycle. Good tools and techniques are needed to assist developers with these tasks. One such technique is data flow analysis. Data flow analysis is a testing technique used to identify anomalies in the sequence of actions performed upon a program’s data elements. Tools based on data flow analysis of procedural programs have been found to be useful in debugging and testing programs.

Objects and object oriented concepts provide an elegant way to model a program’s structures. Using these techniques, a natural solution can be developed for implementing dynamic data flow analysis tools. Some initial work has been undertaken to extend procedural approaches to perform data flow analysis for object oriented programs. These approaches have not taken advantage of object oriented principals, and their solutions tend to be mainly procedural in nature.

The practicality of a dynamic data flow analysis approach is very important to ensure that tools developed for these approaches are capable of being used by developers. Specifically these tools need to be able to work for individual developers testing their contributions within a larger team solution. Many of the existing dynamic data flow analysis techniques do not
provide sufficient details in this regard. Specifically dynamic data flow analysis approaches need to ensure that a targeted analysis is possible, thereby allowing developers to check sections of the code that are relevant to them.

This research aims to develop an approach for performing dynamic data flow analysis for object oriented programs that is both a natural solution, taking advantage of object oriented principals, and is also usable by developers for debugging and testing their programs.

1.2 Contribution and Limitations

This thesis makes several contributions in relation to the furthering of dynamic data flow analysis. These contributions are:

1. A structured approach for evaluating dynamic data flow analysis approaches is presented. This is done by examining the requirements for dynamic data flow analysis, and will allow the evaluation of new and existing approaches.

2. The thesis presents a meta-model based approach for performing dynamic data flow analysis on object oriented programs. A meta-meta-model is provided to indicate the entities that are required within an implementation of this new approach.

3. A meta-model is implemented for the Java programming language. This meta-model provides the capability to analyse Java programs. A prototype implementation of this meta-model is available upon request (email requests to acain@swin.edu.au).

4. Several techniques for performing program instrumentation are presented. The instrumentation of the program allows inserted instructions to report actions to the meta-model. The techniques presented range from compile time instrumentation to runtime monitoring.

5. Implementations of these instrumentation techniques are presented for the Java programming language. The effectiveness of each technique is presented and compared. The implementation of the meta-model includes a number of tests that demonstrate these techniques for the interested reader.

6. The meta-model approach and the instrumentation techniques further the capabilities of dynamic data flow analysis in relation to targeted analysis.
1.3 Outline

This thesis is organized in four parts. Part I contains this introduction and a detailed introduction to dynamic data flow analysis. Part II and Part III contain the main content of the thesis. Part II describes existing approaches to modeling dynamic data flow analysis information and then presents a new, object oriented, approach. Part II concludes with an implementation of the new approach for the Java programming language. In order for the model to perform the required analysis it requires information from a running program. Existing and new approaches for gathering this information are presented in Part III. The thesis is concluded with an overview of the main observations and provides an outline of future directions.
Chapter 2

Principles of Data Flow Analysis

One of the aims of software development is to create quality software artifacts. The quality of a software product has many aspects, one of which is correctness, which refers to the absence of defects. Software testing aims to identify defects that exist within a software product, thereby enabling the development team to implement corrections. There are many different techniques for testing software, one of which is data flow analysis.

Data flow analysis was originally used as a technique for code optimization in compilers [1]. It has also been shown to be a useful technique in other areas, such as performance tuning [15], testing [26, 27], and debugging [24]. This chapter describes the fundamentals of data flow analysis, and specifically dynamic data flow analysis. The chapter concludes with a number of requirements for new testing approaches using dynamic data flow analysis.

2.1 Data Flow in Testing

Data flow information is used in a number of different ways within the testing domain. The main categories are data flow testing, and data flow analysis. Data flow testing refers to the use of data flow information to guide the selection of test cases [26, 27]. In data flow testing, the source of a program is examined to determine paths from the definition to the use of a variable, named def-use pairs. The def-use pairs provide a set of paths, all of which should be covered during program testing [20, 27]. Data flow analysis, on the other hand, is used to detect improper use of data within a program’s source code [23]. Data flow analysis is done by examining the sequences of actions performed upon the data elements within the program, with certain action sequences indicating a potential incorrect use of the data associated with
variables. Anomalies are reported when an invalid sequence of actions is detected for variables under analysis.

2.2 Data Flow Analysis

Osterweil and Fosdick [23] proposed the use of data flow information for the purpose of software validation. This paper outlined the DAVE system, a software validation tool to perform data flow analysis for ANSI Fortran programs, and defined two rules for data flow analysis. These rules define the valid sequences of actions that can be performed on variables within a program. These rules form the basis for all later forms of data flow analysis. The two rules are:

1. A reference must be preceded by an assignment, without an intervening undefined.

2. A definition must be followed by a reference, before another definition or undefined.

In this context, a reference refers to the use of a variable, definition refers to the assignment, and undefined occurs when the variable passes out of scope. These two rules allow the detection of cases where a variable is used prior to being assigned a value, and where a variable is assigned a value that is never used.

Data flow analysis can be performed statically or dynamically: the static approach performs the analysis without executing the program, whereas dynamic analysis is performed by executing an instrumented version of the program. Static analysis is capable of detecting errors within the code but difficulties arise when it is applied to individual array elements, pointers, reference variables, and reference types. Dynamic analysis is capable of analyzing these data types, but is only performed on the parts of the program that are accessed during execution. Dynamic and static analysis are not seen as competing, but as complementary strategies [11].

2.3 Dynamic Data Flow Analysis

Dynamic data flow analysis techniques have been proposed for various procedural [6, 8, 17, 24, 25] and object oriented [2, 9, 10] programming languages. All of these existing techniques follow the basic approach proposed by Huang [16, 17]. Huang’s approach uses program instrumentation to detect data flow anomalies. Using this approach, a program is instrumented with instructions to report the actions that are performed upon the program’s variables. These
additional instructions, called *probes*, are executed when the instrumented version of the program is run. The information gathered from the probes is then used to perform the data flow analysis, and thereby identify anomalies related to the use of the program's variables.

In a number of cases it will not be possible to instrument all of a program. This is likely to occur in cases where third party components or libraries are used, but can also occur for large projects where complete instrumentation is not practical. In these cases we can consider the program as consisting of two parts, the section that contains the probes, being the instrumented code, and the part that does not, named uninstrumented code.

In conventional dynamic data flow analysis [16, 17] there are three basic actions that can be performed upon a variable, namely *define*, *reference*, and *undefined*. A variable is considered to be defined when its value is set; it is referenced when its value is referred to; and is undefined when its value is destroyed or it goes out of scope. Data flow anomalies represent improper sequences of actions performed on a data element, as described by the rules of Osterweil and Fosdick [23].

For example, consider the following C++ code:

```c++
static double PI = 3.14159265;
int main() {
  double x;
  int r;
  x = PI * r * r;
  cout << x;
}
```

Listing 2.1: Data flow analysis example.

Variables PI, x and r are declared on lines 1, 5 and 6. As there is no assignment, there are no actions that need to be reported for these two lines of code. On line 8 four actions occur on these variables: PI is referenced, r is referenced twice, then a define action occurs on x. Line 10 has one reference action occurring on the x variable. Lastly, at the end of the main method, both variables are undefined. Therefore, the sequence of actions performed upon the x variable is *Define - Reference - Undefine*, while the sequence of actions performed on the r variable is *Reference - Reference - Undefined*. From this sequence it is possible to see that variable r has violated the rule 1, ‘A reference must be preceded by an assignment.’ [23]. This would be reported as an anomaly for this code.
With dynamic data flow analysis the two data flow analysis rules [23] are separated into three data flow anomalies; namely, define-define, define-undefine, and undefine-reference. The define-define anomaly indicates that the data element has been assigned a value that has never been used. If a variable that was undefined receives a reference action, this indicates an undefine-reference anomaly. The undefine-reference anomaly is an erroneous sequence, as an undefined data element should not be accessed.\(^1\) The define-undefine anomaly indicates that the data element’s value has been defined but not used before the value is destroyed. The define-undefine, and define-define anomalies do not necessarily indicate the presence of an error.

In order to improve efficiency, Huang [17] proposed the tracking of states via a state machine instead of tracking the sequence of actions. In its basic form, there are four states, namely defined, undefined, referenced, and anomaly. The three actions on variables act as triggers to state transitions and their effects are illustrated in Figure 2.1\(^2\). More advanced state machines are also presented in [2, 5, 6, 8, 9, 10, 17]. These state machines present a number of improvements on the basic variant. Huang [17] presented an alternative which allows continued analysis after anomaly detection. Chan and Chen provide a new diagram for the analysis of parameters across a subprogram boundary [5, 6], while a number of papers presented language specific versions; including Java [2], Pascal [6], Cobol [8], C [24], and C++ [9, 10]. The figure illustrated here is using UML version 1.5 [22], unless otherwise stated, the diagrams in this thesis use this version of UML.

\[\text{Figure 2.1: State chart representing the state machine used for data flow analysis}\]

\(^1\)Static analysis is able to detect these anomalies for variables local to methods, but is unable to do this completely for pointers or instance variables.

\(^2\)It should be noted that the cycles have been removed from these statecharts for simplicity.
2 PRINCIPLES OF DATA FLOW ANALYSIS

2.3.1 Structured Dynamic Data Flow Analysis

Using Huang's approach [17], each variable has an explicitly declared state variable whose name is defined by adding a reserved prefix to the corresponding variable's name. Actions on a variable result in an appropriate transformation on the associated state variable. For example, the code in listing 2.2 is an instrumented version of the listing 2.1. Here the prefix \texttt{s.} has been used to create the state variables. So variable \texttt{s.x} is the state variable for variable \texttt{x}. The \texttt{state_function} is used to perform the transitions described by the state machine.

```c
static double PI = 3.14159265;

int main()
{
    double x;
    state s.x;
    int r;
    state s.r;

    state_function(s.r, Reference);
    state_function(s.r, Reference);
    state_function(s.x, Define);

    x = PI * r * r;

    state_function(s.x, Reference);
    cout << x;

    state_function(s.x, Undefine);

    state_function(s.r, Undefine);
}
```

Listing 2.2: Instrumented data flow analysis example.

Using Huang’s [17] approach, data structures such as arrays and lists have parallel structures containing the state information. This was found to be unsatisfactory for dynamic data structures, such as linked lists [6]. To address this, Chan and Chen [6] proposed that the state information be embedded within the data structure, by modifying the type of the data structure to include both the data elements and their state information. This enables easier analysis of record like structures, though it may not be possible where the program interacts with uninstrumented code.

Both the approaches of Huang [17] and Chan and Chen [6] use state variables linked to their associated data item via name. State variables named in such a way are called \textit{explicit} state variables. Price [24] proposed to use the notion of \textit{implicit} state variables. With this approach, the data element’s \textit{memory location} and \textit{size} are used as the identification keys to track the actions related to a variable. Using this approach, the state information can be stored
in a table separately from the programs data structures. This allows the tracking of dynamic data structures, while better enabling the use of uninstrumented code as variables can be tracked without needing to knows its variable name.

2.3.2 Object Oriented Dynamic Data Flow Analysis

There are two existing approaches for dynamic data flow analysis of object oriented programs, one based on C++, the other on Java. Chen and Low [9] presented a methodology for performing dynamic data flow analysis of C++ programs. This methodology uses implicit state variables, and a number of tables to track and analyse a program. As Java does not provide features that will allow the use of implicit state variables, Boujarwah et al. [2] used a combination of string identifiers to track actions upon variables. Both of these approaches are discussed in further detail in parts II and III. Part II will concentrate on how data flow information is modeled by these approaches, while Part III examines how information is gathered from the instrumented program.

2.4 Requirements for Data Flow Analysis

To effectively evaluate the existing dynamic data flow analysis approaches for object oriented programs, a set of objectives must be defined. These requirements should specify what is required of an approach to enable it to be used in typical software development projects. In [3] we presented a number of requirements that must be met to perform a complete and practical analysis of a program. These requirements are as follows.

1. As a minimum the approach must allow the tracking of actions for the definition, reference and destruction of all variables under investigation.

2. The approach must be able to handle any type of variable, independent of scope, type, or visibility.

3. The approach must support targeted analysis of source, thereby allowing analysis of individual parts of a system, and also allowing analysis of systems that use third party components.

4. The output generated by the approach must enable programmers to identify the location and type of any anomalies produced.
5. The approach must enable automated analysis. This includes the ability to instrument the program under analysis, and generate anomaly report information from the results collected.

In object oriented programming languages, variables are assigned to a particular scope and have an explicit visibility. Hence, any approach must be able to handle global, class, instance, and local variables, as well as method parameters and constants.

The types of variables supported by the language will affect the way analysis is performed. Languages like Java and C# [12] have two different kinds of types. These are referred to as primitive and reference types in the Java language specification [14], and as value and reference types in the CLI standard [13]. Variables of a reference type contain a reference to an object [14] or bit sequence [13]. It is, therefore, possible for one object to be referenced by multiple variables, and thus modifications made via one variable may affect the object referred to by another variable. Variables of a value type directly contain their data [13, 14]. As each value type variable contains its own storage it is not possible for changes to a value type variable to affect the value of another variable. Requirement 2 ensures that correct analysis for both types of variables is supported by the approach.

Because data flow analysis focuses on the data element, not the variable itself, the method used to pass arguments to parameters will affect the parameter’s analysis; an issue that is addressed in Requirement 2. Actions on a parameter passed by value have no effect on the argument that was passed to the parameter. While actions on a parameter passed by reference will directly affect the argument that was passed to the parameter. Table 2.4 shows the possible combinations that result if a language supports both passing by reference and value, and reference and value types, respectively.

Targeted analysis, Requirement 3, refers to the ability to analyze only part of a program. This requirement is of key importance in programs that make use of libraries or third party components. The approach cannot require all code as the source of used libraries and components may not be available.

Individually each of these requirements can be met, it is only by combining these requirements that the complexity of the analysis is highlighted. Targeted analysis, Requirement 3, combined with required analysis of publicly scoped variables, Requirement 2, represents one of these cases (i.e. the entire source may not be available for analysis). Code contained within the unavailable code may use and modify the values of publicly scoped variables exposed from
within the analyzed code. Hence, a lower level of analysis may prove difficult (or even impossible) in relation to the identification of the anomalies within the program’s source.

Of all of the requirements, targeted analysis is the most difficult to meet. This requirement is essential to allowing data flow analysis for large programs, and programs that make use of external libraries for which the source code may not be available. Section 2.4.1 looks at existing approaches to address this issue, and introduces the additional complexities related to targeted analysis for object oriented programs.

### 2.4.1 Approaches for Targeted Analysis

Targeted data flow analysis has two aspects; where an uninstrumented method is called by an instrumented method, and visa versa. This is illustrated in Figure 2.2, a structure chart that shows two procedure calls, Procedure 1 and Procedure 2. Targeted analysis needs to consider the possibility that Procedure 1 is instrumented and Procedure 2 is not, or that Procedure 2 is instrumented and Procedure1 is not. In both cases, the states for the data shared between the methods must be carefully considered.

In proposing dynamic data flow analysis as a testing technique, Huang [17] highlighted the need for targeted analysis. This requirement was further echoed in later papers on data flow analysis for procedural programming languages [6, 11]. There are two main approaches to addressing this issue [5, 17], both of which examine data sharing via parameters.

The first approach was presented by Huang [17], who used the term segmentation of data flow analysis, rather than targeted analysis. Using this approach, the state for a variable is set

<table>
<thead>
<tr>
<th>x is Passed</th>
<th>Reference Type</th>
<th>Value Type</th>
<th>Result of ( y = z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>By Value</td>
<td>( y ) is a copy of the reference contained in ( x )</td>
<td>( y ) is a copy of the value contained in ( x )</td>
<td>has no effect on ( x )</td>
</tr>
<tr>
<td>By Reference</td>
<td>( y ) is a reference to ( x ).</td>
<td>( y ) is a reference to the storage of ( x )</td>
<td>( x ) is assigned the value of ( z ) (value types), or ( x ) is changed to refer to the same location as ( z ) (reference types).</td>
</tr>
</tbody>
</table>

Table 2.1: Method void foo(\( Type \) y) \{ \ldots y = z; \ldots \} is called using foo(\( x \)), where \( Type \) can be replaced with any value or reference type.
Figure 2.2: Targeted Analysis Aspects Illustration

to referenced after the execution of uninstrumented code. For example, let us first consider the case where Procedure 1 is instrumented and Procedure 2 is not. Using the approach presented, the states for the shared data is set to referenced in Procedure 1 after Procedure 2 returns. In the case where Procedure 2 is instrumented and Procedure 1 is not, the state of the shared data is set to referenced as the first action within Procedure 2. The referenced state was selected as this state causes the least number of anomalies, at the risk of not reporting some anomalies that may have occurred.

Chan and Chen [5] proposed a more detailed approach to this issue. Their approach introduces a new set of states for parameter variables; defined and should be referenced named DS, defined for reference only named DR, and reference only named RO. Each of the parameters to a subroutine is placed in one of four possible classifications by the programmer. This additional information can then be used by the dynamic data flow analysis tool to give more detailed results. Table 2.2 lists the classification of the parameter, the state to be used as the initial state within the subroutine, and the action(s) that are used on the state variable of the argument following the call. For example, if the call myRoutine(b) is made where b is categorised as DR, then following the call the state associated with b is referenced as indicated by the post call action.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Initial State</th>
<th>Post Call Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure input parameter</td>
<td>DR</td>
<td>reference</td>
</tr>
<tr>
<td>Input/output parameter</td>
<td>DS</td>
<td>reference and define</td>
</tr>
<tr>
<td>Input parameter used within the subroutine</td>
<td>DS</td>
<td>undefine</td>
</tr>
<tr>
<td>Output parameter</td>
<td>U</td>
<td>define</td>
</tr>
</tbody>
</table>

Table 2.2: Parameter classification table.

Using these classifications makes it possible to perform data flow analysis at an individual
Each subroutine is not dependent on the calling subroutine, nor is it dependent on the subroutines that it calls. To use the approach of Chan and Chen [5], the programmer must classify the parameters within their program. In the *AIDA* system described in [6], the instrumentation process prompts the programmer to enter these details.

Neither of the approaches presented manages to solve the issues related to targeted analysis in an entirely satisfactory manner. The approach of Huang [17] will fail to detect certain anomalies that may occur in the interaction of instrumented and un-instrumented code. Such as the case where the uninstrumented code fails to define a shared data element which is later relied upon by the instrumented code. Many of these issues are addressed by Chan and Chen [5, 6], however, this approach introduces programmer intervention in the instrumentation process, and as a result cannot be fully automated. In addition to these issues, neither of the approaches addresses the issue of targeted analysis and global data.

Object oriented programming languages introduce additional complexities for targeted analysis. These complexities relate to the use of inheritance and polymorphism, and the existence of instance variables. The issues related to inheritance and polymorphism are not considered by the existing approaches as these concepts are not present in procedural languages. In order to provide data flow analysis for object oriented languages these issues must be addressed.

The approach of Huang [17] requires that additional probes be inserted either as the first action within a method called by uninstrumented code, or as the action that follows the call to an uninstrumented method. In the presence of inheritance polymorphism [4], a single call to a method may call instrumented code in some cases, and uninstrumented code in others. This makes it difficult to determine the locations at which these instructions must be inserted. Chan and Chen’s approach [5, 6] is capable of handling individual methods and, therefore, does not suffer from similar complexities. However, the accepted good practice of writing many small methods in object oriented programming, increases the issue of programmer intervention in the instrumentation process.

The greatest obstacle to targeted analysis of object oriented programs is related to the shared data contained within objects, and potentially classes. These variables are ‘globally’ accessible from within the object. Where uninstrumented code exists within these scopes, a

---

Assuming that we ignore global data.
targeted analysis becomes difficult to perform. As global variables are not addressed by the approaches previously presented [5, 6, 17], they do not offer much insight as to how this issue can be solved.

2.5 New Approaches

Having examined the existing approaches for dynamic data flow analysis, two distinct issues can be identified. Firstly, approaches to dynamic data flow analysis provide a means to store and reference state information related to data elements, which is referred to as modeling of state information in this thesis. Secondly, data flow analysis approaches provide a mechanism that allows action information to be extract from the program under analysis, referred to as collecting action information here. Ideally it should be possible to develop the model independently of the mechanism used to collect the action information.

In summary, new approaches to dynamic data flow analysis should meet all of the requirements from section 2.4. In order to do this clearly, new approaches can be divided into two separate concerns.

1. Modeling of data flow information, including a method to identify and locate state variables.

2. A mechanism to interact with the model from the instrumented program, which must address targeted analysis and clearly express any limitations.

The model itself should, as much as possible, be independent of the method used to extract the required information from the program under analysis. In this way targeted analysis becomes a responsibility of the mechanism used to collect action information. In order to concentrate on these issues the body of this thesis is divided into two parts. Part 2 concentrates on the modeling of state information, and presents a new object oriented approach for achieving this. Part 3 provides information on how action information can be collected for a Java program and used by the model presented in Part 2.
Part II

Modeling
Chapter 3

Existing Modeling Approaches

This chapter outlines existing approaches taken to modeling dynamic data flow information for object oriented programs. There are currently only two existing approaches for performing dynamic data flow analysis for object oriented programs. Both of these existing approaches are discussed; the first section discusses the approach for C++, followed by the details of the approach for Java.

3.1 Dynamic Data Flow Analysis for C++

Chen and Low [9, 10] present a methodology for performing dynamic data flow analysis of C++ programs. This methodology provides high level guidelines for the collection, and analysis of data flow information. The following sections outline the steps of the methodology, and introduces its tabular approach to modeling data flow information. The code given in Appendix A.1 will be used to assist this illustration. This code defines two classes, Point and MovablePoint, and two global functions, main and operator<<. In main a MovableParticle is created, moved, then printed to the console using cout.

3.1.1 Overview

Chen and Low [9] model data flow information as a number of tables that store state information about the data elements within the program. Variables themselves are modeled as an area of memory, having a memory location and size. Actions occurring within the area of memory occupied by a variable can then be used to trigger the required state transitions. Using this approach allows the methodology to focus on the data elements being analysed. Three tables...
are constructed in order to track variables by their memory location. These are the variable
register, the Class Access Table, and the Action Table. The methodology itself consists of three
steps. The first, constructs the variable register that maps each variable to its memory location,
the second step creates the Class Access Table that indicates the classes that have access to
each variable, and the third step executes the program, storing the action in the Action Table,
and uses a state machine to perform the analysis on the program’s variables.

3.1.2 The Tables

The first step of the methodology [9] constructs the variable register, a table that maps the
variables in the program to their memory location. The three elements of the variable register
are the variable name, the class in which the variable is defined, and the variable’s memory
location. In the case of global functions the class is set to ‘Global’, an identifier reserved for
this purpose.

The variable register has two purposes; firstly to provide a mapping from memory locations
to variables in the source code, and secondly to map the memory locations to classes. The
mapping of a memory location to a class is used in the second step of this methodology. The
variable register for the sample program is shown in Table 3.1. The class for the variable tnow
is set to Global, as Main is not a member of any class.

The variable register contains all variables other than those that refer directly\(^1\) to objects.
This requirement is not explicitly stated in [9] or [10], but is indicated in their examples. Had
these variables been included in the register, the memory to which they refer would be included
twice. For example, the variable \(p\) in main has the location 0xbffed140-0xbffed14c. This range
is also occupied by \(p\)’s \(x\), \(y\), and \(delta\) instance variables. If \(p\) is included in the register, these
locations will have two entries.

The handling of local variables is not explicitly covered in [9]. The example presented
appears to indicate that local variables take their ‘class’ value for the variable register from the
class in which the method is defined. Following their example, the class name for the \(x\) and
\(y\) parameters in Point’s constructor are set to ‘Point’, while the \(x\), \(y\), and \(delta\) parameters in
MovablePoint’s constructor are set to ‘MovablePoint’.

The second step of the methodology creates a table which indicates the classes that are
permitted to access each variable. This table is referred to as the Class Access Table, and

\(^1\)Pointers to objects are included in the variable register.
contains the memory location of the variable, the classes that are allowed to access it, and the
name of the variable. For the Point example, the Class Access Table is shown in Table 3.2. As
can be seen from the table, the x and y instance variables of the p object are accessible from
the Point and MovablePoint classes, while the delta instance variable is accessible only from
the MovablePoint class. This table also contains rows for the local variables and parameters
of the program.

<table>
<thead>
<tr>
<th>Location</th>
<th>Classes Allowed</th>
<th>Reference Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffed15c-0xbffed160</td>
<td>Global</td>
<td>tnow</td>
</tr>
<tr>
<td>0xbffed114-0xbffed118</td>
<td>Point</td>
<td>x</td>
</tr>
<tr>
<td>0xbffed118-0xbffed11c</td>
<td>Point</td>
<td>y</td>
</tr>
<tr>
<td>0xbffed140-0xbffed144</td>
<td>MovablePoint</td>
<td>p.x</td>
</tr>
<tr>
<td>0xbffed144-0xbffed148</td>
<td>MovablePoint</td>
<td>p.y</td>
</tr>
<tr>
<td>0xbffed148-0xbffed14c</td>
<td>MovablePoint</td>
<td>p.delta</td>
</tr>
<tr>
<td>0xbffed134-0xbffed138</td>
<td>MovablePoint</td>
<td>x</td>
</tr>
<tr>
<td>0xbffed138-0xbffed13c</td>
<td>MovablePoint</td>
<td>y</td>
</tr>
<tr>
<td>0xbffed13c-0xbffed140</td>
<td>MovablePoint</td>
<td>delta</td>
</tr>
</tbody>
</table>

Table 3.2: Class access table.

In step three, the instrumented program is executed, and a state machine is used to analyse
the results. Each action in the program reports the memory location upon which it is operating,
the type of action that was performed, and the class from which the action originated. This
information is recorded in the Action Table, which together with the Class Access Table, are
used to determine the state for each data item. A sample of the data generated by this step
is shown in Table 3.3. This table lists the actions that occur from lines 39 to 49 of the point
sample in listing A.1. The action column lists the type of action, and the originating class, with
the type of action being one of either u for undefine, d for define, or r for reference.
### 3.1.3 Anomaly Detection

Chen and Low [9] propose an alternative to Huang’s state machine [17] (see Figure 2.1) for analysing variables in C++, shown in Figure 3.1. The state machine uses information on the location of the action, and the allowed locations, to raise additional anomalies. These anomalies occur when a data item is accessed from a class that is not included in the variable’s list of allowed classes, as indicated by step two of the methodology. These new anomalies aim to detect cases where data is incorrectly accessed via pointers.

The type of anomalies that can be detected by this methodology are discussed in [10]. This paper identifies a number of anomalies related to the use of classes, pointers and arrays, control flow mechanisms, operators, and variables. While most of the anomalies identified are relevant for any dynamic data flow analysis approach, the methodology does introduce the ability to detect the use of private instance variables from outside the class in which the variable is defined. While it is not possible to directly access these variables, access can be gained via pointers. For example, the code in Listing 3.1 shows a case where a private instance variable of `p` is accessed directly from within `main`. This methodology allows the detection of these cases, using the information from the Class Access Table. In order to handle the new and delete operators, Chen and Low [10] created another state machine specifically designed for

<table>
<thead>
<tr>
<th>Line</th>
<th>Location</th>
<th>Action</th>
<th>Reference Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>0xbfed15c-0xbfed160</td>
<td>u, Global</td>
<td>tnow</td>
</tr>
<tr>
<td>46</td>
<td>0xbfed15c-0xbfed160</td>
<td>r, Global</td>
<td>tnow</td>
</tr>
<tr>
<td>49</td>
<td>0xbfed140-0xbfed144</td>
<td>u, MovablePoint</td>
<td>p.x</td>
</tr>
<tr>
<td>49</td>
<td>0xbfed144-0xbfed148</td>
<td>u, MovablePoint</td>
<td>p.y</td>
</tr>
<tr>
<td>49</td>
<td>0xbfed148-0xbfed14c</td>
<td>u, MovablePoint</td>
<td>p.delta</td>
</tr>
<tr>
<td>7</td>
<td>0xbfed114-0xbfed118</td>
<td>d, Point</td>
<td>x</td>
</tr>
<tr>
<td>7</td>
<td>0xbfed118-0xbfed11c</td>
<td>d, Point</td>
<td>y</td>
</tr>
<tr>
<td>22</td>
<td>0xbfed134-0xbfed138</td>
<td>d, MovablePoint</td>
<td>x</td>
</tr>
<tr>
<td>22</td>
<td>0xbfed138-0xbfed13c</td>
<td>d, MovablePoint</td>
<td>y</td>
</tr>
<tr>
<td>22</td>
<td>0xbfed13c-0xbfed140</td>
<td>d, MovablePoint</td>
<td>delta</td>
</tr>
<tr>
<td>22</td>
<td>0xbfed134-0xbfed138</td>
<td>r, MovablePoint</td>
<td>x</td>
</tr>
<tr>
<td>22</td>
<td>0xbfed138-0xbfed13c</td>
<td>r, MovablePoint</td>
<td>y</td>
</tr>
<tr>
<td>9</td>
<td>0xbfed114-0xbfed118</td>
<td>r, Point</td>
<td>x</td>
</tr>
<tr>
<td>9</td>
<td>0xbfed140-0xbfed144</td>
<td>d, Point</td>
<td>p.x</td>
</tr>
<tr>
<td>10</td>
<td>0xbfed118-0xbfed11c</td>
<td>r, Point</td>
<td>y</td>
</tr>
<tr>
<td>10</td>
<td>0xbfed144-0xbfed148</td>
<td>d, Point</td>
<td>p.y</td>
</tr>
<tr>
<td>24</td>
<td>0xbfed13c-0xbfed140</td>
<td>r, MovablePoint</td>
<td>delta</td>
</tr>
<tr>
<td>24</td>
<td>0xbfed148-0xbfed14c</td>
<td>d, MovablePoint</td>
<td>p.delta</td>
</tr>
</tbody>
</table>

Table 3.3: Partial action table.
analysing pointer variables.

```cpp
int main()
{
    MovablePoint p;
    ...

    void *ptr;
    ptr = &p;
    int *ptr2 = (int*)ptr;

    cout << *ptr2;
}
```

Listing 3.1: Accessing private data in C++

This approach will report a large number of inappropriate anomalies in the case of friend functions, and explicit pointer references. For example, the code in Listing 3.2 shows a modified portion of the Point sample’s source code. The `operator<<` function is global, and so the actions reported within this function indicate that the class of origin is the *Global* scope. The details of the actions reported is shown in Table 3.4. Using the extended state transition diagram presented in [9], the *reference* actions on line 40, place p.x and p.y into the anomaly state, due to transition *'all, disallowed'*. The examples in both papers on data flow analysis for C++ [9, 10] include an overloaded
operator. However, in both cases this global function has not had the correct class of origin, resulting in incorrect analysis. If we follow the logic presented in [9] for the Point example, the actions reported in the operator\(<<\) function would have their class of origin set to Point. This is incorrect, as a friend function is not a member of the class [18]. The reason for marking a function as friend, is to allow these external routines to access private members of the class.

Alternatively, this function may be marked as being part of the Point class due to the friend declaration within the class. In effect the instrumentation recognises that this function is allowed to act as if it were part of the Point class. For the operator\(<<\) example this is acceptable as the probes within this can report their actions as coming from the Point class. However, functions may be friends of multiple classes. For example, the compare function in Listing 3.2 is a friend of both Point and OtherClass. In this case, the compare function cannot be instrumented as coming from the Point class alone, and therefore this alternative is not sufficient.

<table>
<thead>
<tr>
<th>Line</th>
<th>Location</th>
<th>Action</th>
<th>Reference Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>0xbffed140-0xbffed144</td>
<td>r, Global</td>
<td>p.x</td>
</tr>
<tr>
<td>40</td>
<td>0xbffed144-0xbffed148</td>
<td>r, Global</td>
<td>p.y</td>
</tr>
</tbody>
</table>

Table 3.4: Action table for operator\(<<\).

Additional unwanted anomalies will occur when methods intentionally pass pointers to external routines. For example, a sort routine may accept a pointer to a structure to sort. When this routine is used to sort a private instance member, this analysis will report a number of anomalies. Increasing the class’s allowed classes will then require the addition of the special Global scope, resulting in analysis that is equivalent to approaches that do not consider the class.

3.1.4 Evaluation

The methodology of Chen and Low [9, 10] uses a highly data focused approach to the modeling of state information. All variables are identified based on their memory location and size. Objects and instance variables are worked into this methodology via the name entries within the variable register, Class Access Table, and Action Table. By doing this the object structures are bypassed and the data elements themselves become the primary focus.

This methodology works successfully in environments where the memory location of a data element is readily available. As a result this methodology cannot be used in environments
```cpp
class Point
{
    protected:
    int x, y;

    public:
    ...
    
    friend ostream& operator<<(ostream& os, Point& p);
    friend int compare(Point& p, OtherClass& oc);
};

class OtherClass
{
    private:
    ...
    
    public:
    friend int compare(Point& p, OtherClass& oc);
}

ostream& operator<<(ostream& os, Point& p)
{
    return os << "X: " << p.x << " Y: " << p.y << endl;
}

int compare(Point& p, OtherClass& oc)
{
    // access the internals of both oc + p
}

int main()
{
    ...
    cout << p;
}

Listing 3.2: Friend functions in C++.
```
where this is not the case.

### 3.2 Data Flow Analysis For Java

Boujarwah et al. [2] present an approach for performing dynamic data flow analysis for Java programs. Their approach models the data flow information within the program as a set of string identifiers. The following sections present the details of this dynamic data flow analysis approach. Throughout this discussion, the code from Appendix A.2 will be used to illustrate their approach.

#### 3.2.1 Identifying Variables

In C++, data flow actions are associated with data elements via their implicit variable name [10]. As Java provides no mechanism to directly determine the memory location of a variable, this approach can not be used. Boujarwah et al. [2] proposed to use an *explicit* data name instead. The explicit data name is a string constructed from the name of the variable, and identifiers related to its location in the program. In the following paragraphs we will describe these identifiers for the different types of variables considered.

The location of a local variable relates to the method in which it is defined and the class in which this method is defined. This results in an identifier containing the variable’s name, the name of the class that contains the method, the name of the method that contains the variable, and the method’s argument types. This ensures that local variables of the same name within overloaded methods remain uniquely identifiable. The variable ‘s’ declared on line 31 of the Particle class from Appendix A.2 is identified by the following values:

1. The name of the variables, s.
2. The name of the class, Particle.
3. The name of the method, toString.
4. The name of the argument types, () i.e. none in this case.

This information is used to uniquely identify local variables when they are used within a program under analysis. The data flow actions related to s from the above sample will identify
the variable as ‘s, Particle, toString, ()’. These identifiers are used by the analysis system to associate actions with the respective variable. Similarly, the args parameter in the main method on line 38 is identified with ‘args, ParticleTest, main, (String[])’. Boujarwah et al. [2] do not include the package name of the class in their type identifiers, though this may be required in certain circumstances.

Instance fields are located within objects. To identify instance fields requires a mechanism to identify objects. In [2] the location of an instance field is described using an object identifier, the name of the instance field, and the name of the class in which the field is defined. Each object is identified by its ‘object name’, the name of the method in which the object was created, the types of the arguments for this method, and the name of the class in which the method was declared. The term ‘object name’ is not explicitly defined in [2], though the illustrations provided indicate that this is the name of the variable used to refer to the object when it was created. From our example the fields of the Particle instance created on line 40 are identified by the following values:

1. The name of the instance field, x or y.
2. The object identifier, consisting of
   (a) The object name, p.
   (b) The method name, main.
   (c) The method’s arguments, (String[]).
   (d) The name of the class containing the method, ParticleTest.
3. The name of the class in which the field is defined, Particle in both cases.

The analysis system uses this information to uniquely identify instance fields. The data flow actions reported for the field x of the object discussed above would identify the variable using ‘x, p, main, (String[]), ParticleTest, Particle’.

The identification of static fields is performed using the name of the field and the name of the class in which it is defined. The rng variable declared on line 5 of Listing A.2 is a static field that it identified using the information provided in the following list.

1. The name of the static field, rng.
2. The name of the class in which the field is defined, \texttt{Particle}.

When \texttt{rng} is used or modified, the variable is identified by `\texttt{rng, Particle}`.

Method return values are identified in a similar way to instance fields. Using the approach presented, the method is identified by the object on which it is executed, the method’s name, its argument types, and the class in which the method is declared. No details are given for identifying return values from static methods, though the approach for identifying static fields may be able to be extended to enable this. In the \texttt{Particle} example the return value from the \texttt{getX} method call on line 44 is identified as shown below.

1. The object identifier, consisting of
   (a) The object name, \texttt{p}.
   (b) The method name, \texttt{main}.
   (c) The method’s arguments, \texttt{String[]}.
   (d) The name of the class containing the method, \texttt{ParticleTest}.

2. The name of the method, \texttt{getX}.

3. The method’s arguments, ()

4. The name of the class containing the method, \texttt{Particle}.

This information is used to uniquely identify data flows related to method return values.

When these actions occur for the \texttt{getX} method, from the above example, it is identified by `\texttt{p, main, (String[]), ParticleTest, getX, (), Particle}`.

Reference variables are treated specially in [2]. These variables are excluded from the analysis, and are used exclusively for the identification of objects. When a reference variable is assigned to another, such as is shown on line 43 of Figure \texttt{A.2}, bookkeeping is performed to indicate that `\texttt{variable1 = variable2}`. For the example on line 43, \texttt{p1} is now an alias of the object name \texttt{p}. In order to ensure that the model is updated, this information is reported as `\texttt{p1 = p}`. From this point on, actions reported for the \texttt{p1} variable will be directed to the \texttt{p} object.

When actions are reported for the current object (i.e. the object referred to by the pseudo variable \texttt{this}) the probes require the object’s identification information. This information includes the \texttt{object name}, method name, arguments and containing class, all of which exist external to the object. For example, when the \texttt{move} method is called, the \texttt{Particle}’s \texttt{x} and \texttt{y} instance
fields are used, see lines 23 – 27 in Listing A.2. When this method is called from main on line 41, the object is identified by ‘p, main, String[], ParticleTest ’, this information must be made available within the move method. Boujarwah et al. [2] address this by introducing a class to manage this information, called the determineObject class. This class contains static fields for each of the elements required for object identification. For the move call on line 41, the determineObject would contain the following information:

1. The object name, p.
2. The method name, main.
3. The method arguments, String[].
4. The class name of the class containing the method, ParticleTest.

All of this information is then used by the analysis system to identify the current object when instance fields are accessed, and for identifying return values from instance methods.

### 3.2.2 Analysis

Boujarwah et.al [2] present a modified version of the Huang’s state machine [17]. In this machine, shown in Figure 3.2, they introduce a new state named Implicitly Defined. This state is used as the initial state for instance variables which are shadowed by instance variables declared by subclasses. This state was introduced to avoid reporting anomalies where these shadowed instance variables are defined during construction and then, without being references, are undefined during object destruction.

For example, the x instance field in the SuperParticle class of Listing 3.3, is shadowed by the instance with the same name in the Particle class. When a Particle is created, the x field from the SuperParticle class is placed in the Implicitly Defined state, and no anomalies are reported for this field if it is not used before the object is destroyed.

The failure to report this anomaly is justified by the argument that “normally, we access the override variable” [2]. However, with Java this is not the case. Java is statically typed and, therefore, all references to the field x via SuperParticle references, will be directed to the field within the SuperParticle class. Additionally, the this pseudo variable within the SuperParticle class is of type SuperParticle, regardless of the object’s type, and therefore all references to this.x in the SuperParticle class will refer to the x field declared in the SuperParticle class.
### Existing Modeling Approaches

```java
public class SuperParticle {
    public int x = 0;
    ...
}

class Particle extends SuperParticle {
    public int x = 10;
    ...
}
```

Listing 3.3: Implicitly Defined illustration

---

Figure 3.2: Extended state chart
3.2.3 Instrumentation

Lastly Boujarwah et al. [2] presented a number of steps to allow the instrumentation of Java programs. The first of the three steps constructs three tables to store the information related to the location of data within the program. The first of these tables is the *instance/class variable table*, and stores information related to the static and instance fields within the program. The table stores the name of the variable, its *usage type* being either instance or class, and the action performed upon the variable during construction. The second table is called the *Inheritance Table*, and contains information that links classes to their immediate super class, and implemented interfaces. Table three is the *Object Table* containing information on the objects constructed within the program. This information is used in determining the object name when an instance field is accessed in the program.

The second step introduces a process for the instrumentation of the Java program using the tables from step one. In this step, probes are inserted into the source code of the program under analysis. The location of the probes and information contained within them are determined as per the previous discussion. The information from the tables constructed from step one, allow the details for the probes to be populated. The instance/class variable table, combined with the inheritance table, allow the probes for object construction and destruction to be populated. The object table combined with the inheritance table provides the details for probes related to the access of instance fields. The result of this step is a newly created Java program that contains the original with instrumented probes. The final step executes the instrumented program, and analyses the results using the new state transition diagram.

Data flow analysis for concurrent Java programs is discussed by Saleh et.al [28]. The paper extends the approach proposed by Boujarwah et al. [2], to enable the analysis of concurrent programs. This makes no changes to the approach in [2], rather it presents a data flow like analysis of the operations that can be performed on a *Thread* object. Hence, it does not provide any further insights into modeling data flow information for Java.

3.2.4 Evaluation

This approach to modeling data flow information is highly error prone. The tracking of local variables via the mentioned string identifiers must be managed very carefully when recursive methods are used. There are also significant challenges related to tracking objects via their *ob-
ject name. It is common for an object to remain within the application longer than the variable through which it was first referenced. If the method in which the object was created is executed again this will result in two objects with the same object name. As a result, this approach to modeling data flow information is insufficient in identifying state information associated with data elements.

### 3.3 Summary

The approaches of Chen and Low [10], and Boujarwah et al. [2] are highly data focused and make little use of object oriented principals. The methodology of Chen and Low [10] uses a number of tables to map memory locations to variable names and allowed classes. The approach of Boujarwah et al. uses a number of string identifiers from the underlying program to model action information.

The methodology of Chen and Low [10] provides sufficient information to enable the modeling of data flow information for C++ programs. The information is modeled as a set of related tables, and does not make use of objects or classes. While this methodology is highly data focused, it is capable of performing dynamic data flow analysis for C++ programs. Targeted analysis is not addressed at this level and will be discussed in Part III of this thesis.

The approach of Boujarwah et al. models data flow information as a set of actions identified by a combination of program identifiers. These identifiers require particular attention and significant management to ensure that they remain valid during program execution. The approach proposed for managing these identifiers suffers from a number of shortcomings that render it unusable in a practical setting.

In general, neither of these two approaches uses object orientation to model the data flow information. As both of these approaches are working with object oriented programs this causes a certain degree of mismatch. In order to address this mismatch, both approaches introduce a level of management to transfer information between the data oriented and object oriented views. An object oriented approach should provide a more natural solution in this environment. The next chapter presents a new approach to dynamic data flow analysis that uses an object oriented model for data flow information.
Chapter 4

Object Oriented Data Flow Analysis

Section 2.4 identified several requirements for dynamic data flow analysis approaches. These requirements were then divided into two separate concerns, modeling of data flow information, and a method to interact with the model from the instrumented program. This chapter proposes a new method for modeling state information for dynamic data flow analysis of object oriented programs.

4.1 Meta Model Approach

Existing approaches to dynamic data flow analysis of object oriented programs have addressed the associated issues without making use of object oriented constructs. In order to provide a more natural solution, the following approach uses object oriented techniques to address these issues.

The first issue to be addressed by a new approach relates to the storing and locating of state information. Using this approach, the states for variables within the running program are stored and located within a meta-model [19] of the program’s structure. This meta-model describes the program’s runtime structure, and contains objects to manage the states for the variables within the program. The meta-model consists of classes that are used to track the actions on variables during program execution. Services of this meta-model are used whenever corresponding actions occur during program execution. As the runtime structure of a program is language dependent, each object oriented language will require a different model. The different models will cater for the specific details of that language and any language dependent features, such as pointers in C/C++ [29], or dynamic scoping in CLOS [33].
The primary responsibility of the meta-model is to observe the actions that are performed upon the variables within the instrumented program. Each variable in the program has an associated object in the meta-model (that is a meta object) which does not only store information to uniquely identify the variable, but also keeps track of all actions performed upon it. For efficiency reasons, this may be done using the state machine based technique introduced by Huang [17]. Since the corresponding actions differ from language to language, the exact states and transitions have to be defined separately for each language. The information stored in the meta objects can later be used to report detected anomalies.

The program meta-models can be described by a meta-meta-model. The meta-meta-model contains classes that define the behaviour and relationships of elements within the meta-model. The meta-meta-model is shown in Figure 4.1. The State Identifiers within the meta-model represent variables, and have the ability to store and manipulate state information for a variable within the program. State Identifiers are located within Scoping Components. Examples of Scoping Components include blocks, methods, objects, and classes. These structures potentially contain a number variables, and define rules for locating them within their context. Scoping Component Containers are collections of Scoping Components, and would include elements such as the method stack and object heap space.

![Figure 4.1: Meta Meta Model](image)

In order to provide the functionality to locate a variable the meta-meta-model provides a mapping between scoping components. When an action is reported to the current scoping component it attempts to locate a matching variable within its context. When this search fails, the scoping component delegates the search to another scoping component based on the rules of the associated language. For example, if a variable \( x \) is defined, the meta-model is informed
and searches for $x$ within the current context. This would typically require looking for an $x$ variable within the current method, then the current object, and so on until the location of $x$ is determined. Once the meta-model has located the meta-object for $x$, it is informed of the action and can update its internal state appropriately. The exact nature of the search for $x$ is determined by the scoping rules of the language for which the model is implemented.

In order for this to perform successfully, the meta-model must contain classes that act as entry points for this searching. To this end, the meta-meta-model contains the *Scoping Component Containers*. These structures are designed as collections of *Scoping Components* that allow searching in particular ways. Implementation of these in the meta-model would reflect elements such as the method stack, and object heap space as mentioned before. Each of these *Scoping Component Containers* would provide access to its *Scoping Components*; the heap for example would allow direct access to a *Scoping Component* based upon its reference while the method stack would only allow access to the current method.

The meta-model relies on appropriate notifications from the program under analysis in order to create a correct meta-level representation of the program. It requires notification of (i) the creation and destruction of scoping components and (ii) the *define*, *reference*, and *un-define* actions for variables. The mechanisms used to obtain these notifications from a running program are independent of the meta-model - an implementation thereof must provide an interface through which notifications can be passed to the model.

This chapter illustrates the steps required to define an implementation of the meta-meta-model. The following chapters will present an illustration of this process for the Java programming language.

### 4.2 Implementing the Technique

The implementation of this technique requires the construction of the language specific meta-model from the meta-meta-model. The meta-model must be adapted to perform dynamic data flow analysis for programs written in the language it was created for. To implement this technique for a language, the following steps must be performed:

1. Define a class, or hierarchy of classes, to perform variable analysis from a sequence of actions, typically performed using a state machine.

2. Create a model that contains the language’s scoping structures.
3. Implement variable searching functionality within the model using the language’s scoping rules.

4. Provide a mechanism to allow the instrumented program to interact with the model.

Each of these steps is described in detail in the following sections.

4.2.1 Variable Analysis

The first step towards implementing this technique for a given language is to define how variables for that language are analysed. Traditionally this has been done using a state transition diagram, as defined in [17]. The approach presented here allows the implementer to define their own analysis method that may, or may not, use state transition diagrams. This provides sufficient flexibility to implement any of the state transition diagrams presented in [2, 6, 8, 17, 25], or to be adapted to new methods of analysis.

This step requires the definition of a set of events and messages related to variable use. When the event occurs within the program the associated message will be passed to a State Identifier within the meta-model. The State Identifier can respond to this message as required. Using the approach of Huang [17], we can define the following set of events and messages, as shown in Table 4.1. The messages are used inside the State Identifier objects to transition between the states Defined, Referenced, Undefined, and Anomaly, as shown in the state chart in Figure 2.1.

<table>
<thead>
<tr>
<th>Event</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>The setting of a variables value</td>
<td>Define</td>
</tr>
<tr>
<td>The reading of a variables value</td>
<td>Reference</td>
</tr>
<tr>
<td>The variable is no longer accessible</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

Table 4.1: Events and Messages

The State Identifier can be implemented as a Variable class, or hierarchy of classes, within the meta-model. An instance of one of these classes contains analysis information for a single variable. When an action is performed upon a variable in the instrumented program, its Variable object in the meta-model is informed of the type action, for example reference, define, or undefine. The Variable object is then able to perform any internal processing to perform the data flow analysis.

---

1The remainder of this chapter will use these events and messages for illustrations. Implementation of this approach are expected to provide their own events and messages.
The State Identifier within the meta-meta-model allows for the presence of arrays and pointers via the delegates to relationship. In the cases where the pointer is dereferenced, the State Identifier of the pointer can delegate the action to an associated State Identifier. For example, the code in Listing 4.1 shows two variables, the integer x and the pointer p. Within the meta-model, the State Identifier for p contains a reference to the State Identifier for x. Line 5 of this program dereferences the pointer p, and assigns a value to the associated location. Within the meta-model this will result in p’s State Identifier being referenced and a define message being sent to x’s State Identifier.

4.2.2 Modeling Scoping Structures

The Scoping Components from the meta-meta-model are used to represent scoping components of the language, such as classes, objects, methods, and blocks. These elements within the meta-model must reflect their language counterparts as both containers for variables, and via their delegation structure. This is implemented in the meta-meta-model via the contains and delegates to relationships. For example, in Java, a block may contain a number of variables, when the variable is not located within the block, it delegates the search to its parent block or method.

The Scoping Components cater for both data structures and call stacks. A meta-model implementation can contain classes to represent the methods on the program’s call stack. Method objects contain appropriate parameter objects, local variables, and will typically delegate to either an object, class, or global scoping component. Objects and instance variables are also represented in the meta-model, with each object under analysis in the instrumented program having an associated representation in the meta-model. Each object’s meta-model element contains a collection of the object’s instance variables. Classes and global variables are also represented in the meta-model with corresponding Scoping Components. The reader should note that targeted analysis is an integral part of this approach and, therefore, objects currently

```c
1 int main ()
2 {
3     int x = 0;
4     int *p = &x;
5     *p = 10;
6 }
```

Listing 4.1: Pointer State Identifiers.
not under analysis do not generate any additional overhead at runtime.

The Scoping Component Containers need to be implemented to control access to the Scoping Components. These elements would map to larger structures within the language such as the method call stack and object heap space. The implementation of these elements must enable the Scoping Components to be retrieved from the container based upon the features of the structure in the language. For example, the heap should allow Scoping Components to be retrieved via its reference.

The meta-model is used as an abstraction, or observation, of the current state of the instrumented program. This representation is used to locate variables, as described in the next section, and to report anomalies. The elements within the model should provide sufficient information to assist developers in tracking down any reported anomalies.

4.2.3 Searching the Model

When an action on a variable occurs, the associated message must be passed to the corresponding State Identifier. This process involves locating the correct object within the meta-model and can be performed either by replicating the language’s scoping rules within the meta-model, or by using detailed runtime information from the program.

Where the language’s scoping rules are implemented within the meta-model, the classes developed must include the ability to search for a variable. This searching functionality uses the representation of the model with the language’s scoping rules. This requires only minimal information from the instrumented program, with the additional cost of subsequent search for the variable within the meta-model. Listing 4.2 illustrates a potential Java implementation of this where source code instrumentation is used. In this illustration, the define method of the DFA class provides the indication of the type of action, and the string ‘x’ indicates the variable upon which the action occurred. When the define message is received by the DFA class, it passes the message to the current Scoping Component, representing the Point constructor. This Scoping Component then searches its collection of variables for x. As no variable is found, the request is passed to the object upon which the method has been executed. The object searches its instance variable collection to identify a match, and will process the action using the identified State Identifier object. If a match had not been found, this process would continue, searching further Scoping Components (such as the object’s class and global variables) until a match was

\footnote{In this illustration block level scoping components are ignored for simplicity}
The way the meta-model handles the situation where no match is identified is dependent on the language being modeled. In languages that require variables to be declared before use, such as C# and Java, this situation never occurs. In this case, the missing variable indicates a deficiency in the model or implementation of the scoping rules. In languages that do not require variable declaration, such as Python and Visual Basic, a missing variable indicates the creation of a new variable within the current scoping component.

An alternative approach is to have more complex probes that provide more information related to the location of the variable. For example in C++, the memory location of the variable upon which the action was performed can be used. This additional information can then be used by the model to locate the variable in a different and possibly more efficient way. In these cases the model is used for reporting, and storage of state information.

4.2.4 Program to Model Interaction

The program under analysis must communicate certain events to the meta-model to allow for the creation of a corresponding representation. These events must include the following:

1. Creation of new, and destruction of existing, Scoping Components including, for example:

   (a) Creation and destruction of objects\(^4\).

   (b) Start and end of methods.

---

\(^3\) If *Option Explicit* is not enabled

\(^4\) This includes the loading of class objects in languages like Java and C#.
(c) Start and end of blocks.

2. Creation of variables.

3. Messages for all actions described in Section 4.2.1 Variable Analysis.

The creation or start of a new scoping component is used by the meta-model to construct an object used to represent this scoping component. When a variable declaration is reported, a new State Identifier meta-object is constructed and added to the appropriate Scoping Component. The messages defined in Section 4.2.1, must be reported to enable the analysis, as previously discussed. The destruction of an existing scoping component is used to remove the matching element from the model. For example when a method terminates, its representation must be removed from the model’s method stack, returning the control to the calling method.

The process used to communicate the events to the meta-model must ensure that only events that actually do occur are reported. This is of particular importance in the context of short-circuit evaluation of expressions and exceptions. An ideal implementation would perform the action and its reporting as an atomic operation. Where the action and reporting cannot be performed atomically (e.g., with source code instrumentation), an implementation must ensure that actions are reported correctly.

4.3 Implementation Requirements

In order to fulfill the requirements presented in section 2.4, an implementation of the meta-meta-model must meet certain criteria. These criteria relate both to the implementation of the meta-meta-model, and to the communication of events from the program under analysis to the meta-model. These criteria are detailed in the following two sections.

4.3.1 Meta-Model Requirements

An implementation of the meta-meta-model must meet the following requirements.

1. State Identifiers must be defined for all variable types in the language.

2. All Scoping Components of the language must be implemented.

3. The meta-model must provide sufficient information for the programmer to identify the location of an anomaly.
4. The meta-model must work in the presence of targeted analysis.

5. Where the meta-model will perform the search for a variable:

   (a) The searching of the model must use the rules of the language to locate \textit{State Identifiers}.

   (b) The relationships between the \textit{Scoping Components} must be constructed to enable this search.

\subsection*{4.3.2 Communication Requirements}

The communication strategy used to communicate the events from the program under analysis to the meta-model, must meet the following requirements:

1. Targeted analysis must be addressed.

2. The implementation must ensure that only events that occur are communicated to the meta-model.

3. All events for the variables within the targeted scope must be communicated to the meta-model.

4. The communication method used must not change the semantics of the program, i.e. the tested program is observably equivalent \cite{7} to the original program.
Chapter 5

Java Model Implementation

This chapter illustrates a simplified implementation of the meta-meta-model presented in Chapter 4 for the Java programming language. This language was chosen because it is object oriented, supports only pass by value, and provides abstractions for runtime introspection. To simplify this illustration, inner and anonymous classes are ignored as well as threads and concurrency. This allows the illustration to concentrate on the process of applying the technique rather than on dealing with the language specific details.

5.1 Variable Analysis for Java

For the purpose of illustration, this implementation will use the basic state machine defined by Huang [17], as shown in Figure 2.1. The events and messages for this implementation are shown in Table 4.1, listed for convenience in Table 5.1. The set of states for the Java implementation includes undefined, defined, referenced, and anomaly. The transitions within this state machine are triggered by the messages define, reference, and undefined, respectively.

<table>
<thead>
<tr>
<th>Event</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>The setting of a variables value</td>
<td>Define</td>
</tr>
<tr>
<td>The reading of a variables value</td>
<td>Reference</td>
</tr>
<tr>
<td>The variable is no longer accessible</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

Table 5.1: Events and Messages for Java

The Java implementation contains only one State Identifier, implemented in the DFA.Variable class illustrated in Figure 5.1. This class is used to represent all variables, regardless of their

\(^1\)Arrays are modeled as a type of Scoping Component.
The DFA Variable class contains the logic to perform the analysis for a single variable.\footnote{The discussion will ignore Java’s default initialization of fields, and all variables will be marked as un-defined until explicitly defined within the program. This will allow for a more detailed discussion of variable analysis that will apply for languages that do not support default initialization.}

The state within the DFA Variable is managed as an integer, and transitions modify this value between designated constants for the states defined, undefined, referenced, and anomaly. The transitions are implemented within the processAction method. The processAction method receives parameters indicating the action that was performed and the method in which the action occurred. The actionType parameter is used to determine the transition, while the inMethod parameter is used for reporting when the action results in a transition into the Anomaly state. When an anomaly occurs, the reportAnomaly method is executed and informed of the method with which the anomaly is associated. The reportAnomaly method uses information from within the meta-model to report the type of anomaly that has occurred and its location, and hence to help developers identify the cause of the anomaly.

The implementation of the DFA Variable class is described in Appendix B, Table B.1, on page 114. This appendix also contains details for all of the other classes discussed in this chapter.

### 5.2 Scoping Structures for Java

There are four categories of variables in Java [14], namely local variables, instance fields, static fields, and interface constants. Local variables, which include method parameters, have a block scope and can be accessed only from the current block and any child blocks. Instance fields can be accessed from within their object or via the object’s reference. Access to instance fields is controlled by a scope modifier: being either public, protected, or private, or default when no explicit modifier is given. Static fields can be accessed from any object created from the class.
in which they are declared, any child class, and also via a reference to the declaring class itself. Access to static fields is also controlled by scope modifiers in the same way as for instance fields. Constants defined within an interface can be accessed in the same fashion as a static field, and are also accessible from any class which implements the interface. To allow the model to represent these separate scopes, classes must be defined to handle these constructs.

Java also includes arrays which are reference types containing an indexed set of values. The values within the array must be considered as individual variables and therefore must be modeled appropriately within the meta-model.

Each of these scoping structures is implemented as a *Scoping Component* from the meta-meta-model. The Java meta-model implements the following classes to support these scoping structures.

1. DFA_Object
2. DFA_Class
3. DFA_Method
4. DFA_StaticInitializer
5. DFA_Block
6. DFA_Array

Each of these classes is described in detail in the following sections.

### 5.2.1 Objects

The DFA_Object class acts as a container for instance fields in the meta-model. The DFA_Object class models a single object and its data during runtime. When an object is created during program execution, a DFA_Object is constructed and used to represent that object for data flow analysis purposes. When a DFA_Object is constructed, it creates DFA_Variable objects to represent its instance variables, which include the instance variables declared in the object’s class, and all parent classes. The DFA_Variable objects are stored within the DFA_Object, and are indexed through a combination of the variable’s name, and the class name which contains the declaration of the variable, similar to the approach taken by Boujarwah et al. [2]. This allows
for the model to handle variable shadowing, where the same variable name is declared in a
class and one or more of its parent classes and interfaces.

Figure 5.2 shows the class diagram for the DFA_Object class. When the DFA_Object
is constructed, the constructor is called and Java’s runtime introspection services are used to
query the class from which the object was created. Using java.lang.Class's getDeclared-
Fields method, the model can determine all of the variables associated with the object. This
information is then used to create DFA_Variable objects that are added to the DFA_Object’s
instanceVariables collection using the addInstanceVariable method. The variables collection
is implemented as a hash table with the key to the variable being a combination of the
class and variable names as described above. Each of the DFA_Variable objects will have a
reference back to the DFA_Object. This association is used for reporting anomalies uncovered
by the DFA_Variable.

Object destruction in Java is only ever triggered by the garbage collector, and program ter-
mination. Just prior to destroying an object, the garbage collector calls the finalize method
on the object. This method can be used to inform the model that the object has been de-
stroyed. Program termination can also be monitored using the Runtime’s addShutdownHook
method [30]. Both of these events trigger the DFA_Object’s destroyed method, and remove
the DFA_Object from the model. Prior to being removed from the model, the destroyed method performs undefined actions on all of the object’s instance variables.

5.2.2 Classes and Interfaces

The instances of the DFA_Class are used to track static fields. At runtime the class is treated as a DFA_Object that contains only the variables declared as static fields, i.e. those that are declared using the static keyword. The DFA_Class inherits its behaviour from the DFA_Object class as shown in Figure 5.3. The DFA_Object’s behaviour is modified so that only the static variables from the monitored class are added to the instance variable collection. DFA_Class objects are created when the Java virtual machine loads a class of the instrumented program, and are destroyed at the end of the program.

The DFA_Object and DFA_Class are Scoping Components that delegate searching based upon instantiation and inheritance. This is implemented within DFA_Object and DFA_Class via the parent relationship. Since each object in Java is an instance of a single class, this information also needs to be stored in the meta-model. For example, the DFA_Object instance associated with an object of a class MyClass contains a reference to the DFA_Class object representing the MyClass class. Furthermore, the parent field in a DFA_Class instance refers to the DFA_Class instance of its direct super class. In the case of java.lang.Object, this reference is null.

A class in Java may also implement a number of interfaces. As the functionalities for interfaces and classes is identical for the purposes of data flow analysis, they are both encoded within the same class DFA_Class. Hence, each DFA_Class object may reference multiple DFA_Class objects via the interfaces field.
5.2.3 Arrays

Arrays in Java are reference types and are managed using a child class of DFA_Object, called DFA_Array, as shown in Figure 5.4. The DFA_Array is a Scoping Component that delegates actions to its variables based upon the index used when accessing the array. When an array is encountered in the instrumented program, the meta-model is informed and a DFA_Array object is created. During the creation process, the DFA_Array creates DFA_Variable objects for each of its elements, and stores them in its elements collection. The elements reference within the DFA_Array duplicates the instance variables reference from DFA_Object. Consequently the instance variables reference within the DFA_Array is not used.

The DFA_Array is only used for the array itself, the references to the array are managed separately as DFA_Variable objects. For example, the source code in Listing 5.1 illustrates the declaration, construction, and use of an array x. The variable x on line 5 of the code will be represented within the meta-model as an instance of DFA_Variable, at this point there is no DFA_Array object related to this variable. Line 6 causes a define action on the DFA_Variable, and the creation of an instance of DFA_Array with two elements. When the array is used on line 7, the DFA_Variable receives a referenced message. The print method’s array parameter will be a new DFA_Variable, however, the actions performed upon the elements of the array will act upon the same array as referred to by x.

5.2.4 Methods and Blocks

Methods within the meta-model are implemented as Scoping Components that delegate variable searches initially to the method’s blocks, and then to the object or class upon which the method
```java
class ArrayTest {
    public static void main(String[] args) {
        int x[];
        x = new int[2];
        print(x);
    }

    public static void print(int[] array) {
        ...
    }
}
```

Listing 5.1: Array Sample Code

was executed. In order to implement this, DFA.Method objects are created and associated with DFA.Block and DFA.Object instances.

Java allows block level variable declarations. This is modeled by the DFA.Method object containing a reference to the current block, a DFA.Block object, as shown in Figure 5.5. Within the method, each block is identified by a unique number. This is used when an anomaly is reported, and for exception handling as described later. When a local variable declaration is encountered in the instrumented program, the addVariable method is executed on the DFA.Method object, which in turn passes the request to the current block. The DFA.Block’s addVariable method creates a new DFA.Variable and adds it to its variables collection.

The DFA.Method is also associated with a DFA.Object instance, representing the object upon which it was executed. Where the method is declared as static, using the static keyword, the DFA.Object will be the appropriate DFA.Class instance. This DFA.Object is required for reporting and to allow the model to access instance, and class, fields that are used within the method.

The DFA.Method class also contains methods to add and remove blocks. When the method object is constructed, a DFA.Block is created and referenced as the method’s current block. When new blocks are entered, the instrumented program uses the DFA.Method.addBlock method to create a new DFA.Block. The new DFA.Block’s previousBlock is set to the method’s current block, and the new block is set as the DFA.Method’s current block. When a block ends, the instrumented program uses the removeBlock method to call the endBlock on the the DFA.Method’s current block, and to set the DFA.Method’s current block to the
Figure 5.5: Method and Block Scoping Components
previous block. The endBlock method results in an *undefine* action being reported to all variables added within that block. The DFA.Method.catchFinallyBlock method is used to handle exceptions.

The model must support Java’s exception handling mechanisms. Therefore, the reporting of the methods termination, via methodEnded, must occur regardless of the way in which the method ended. When an exception is caught, the catchFinallyBlock method uses the block identifier to ensure that the appropriate blocks are ended within the model. This method is also used to handle finally blocks which require similar processing in the case of an exception.

### 5.2.5 Static Initializers

A class in Java may have a number of static initializers. A static initializer is a special method that is executed when the class is loaded by the virtual machine. A static initializer differs from a standard method by the fact that it has no method name or parameters. This is supported in the model by the DFA_StaticInitializer class. This class is shown in Figure 5.6.

### 5.2.6 Java Model

Figure 5.7 presents an overview of the various data flow analysis classes as presented in previous sections. This diagram contains two Class nodes to aid with layout. In both cases this refers to the class java.lang.Class. This structure implements the scoping rules for Java as defined below:

1. Variables are either:
   
   (a) Instance fields, accessed via a DFA_Object object,

   (b) Static fields, accessed via the DFA_Object super class of a DFA_Class object,

   (c) Interface constants, accessed via a DFA_Class’s interfaces,
5 JAVA MODEL IMPLEMENTATION

Figure 5.7: Data flow analysis classes

(d) Array Elements, contained within a DFA_Array, or

(e) Local to methods, contained within the DFA_Blocks of a DFA_Method.

2. An object is created from a class, implemented via the parent relationship between DFA_Object and DFA_Class.

3. A class has a single parent class, or no parent in the case of java.lang.Object, also implemented via the parent relationship between DFA_Object and DFA_Class.

4. A class may implement a number of interfaces.

5. A method is executed on an object or class.

6. A method has a current block, which in turn has previous blocks.

The next section examines the scoping component containers within the model.

5.3 Scoping Component Containers

The main Scoping Components of the Java meta-model are DFA_Method and DFA_Object. DFA_Method provides access to local variables declared within the methods blocks, while
DFA\_Object provides the functionality for arrays, fields, and interface constants. Access to objects of these two classes is provided via *Scoping Component Containers* representing the method stack and heap space. These are illustrated in Figure 5.8.

![Figure 5.8: Heap and Stack](image)

### 5.3.1 Heap Space

The *HeapSpace* class maintains a collection of DFA\_Objects. This class is responsible for locating and creating DFA\_Objects. The `getDFAObjectFor` method implements these main responsibilities. This method initially attempts to locate the matching DFA\_Object within the collection, and returns it if found. If no object is found the object must exist outside of the scope of the targeted analysis, and is therefore ignored. The methods of *HeapSpace* are illustrated in Figure 5.9.

### 5.3.2 Method Stack

In a similar fashion to the *HeapSpace*, the *MethodStack* class\(^3\) is a *Scoping Component Container*. The *MethodStack* class maintains a stack of DFA\_Method objects, with the currently

\(^3\)This class could also be named DFA\_Thread. As this model is not designed for concurrent environments the name *MethodStack* was chosen to avoid confusion.

![Figure 5.9: Heap Space class](image)
executing method on top. The class provides operations to add new methods to the stack, and to manage the blocks within the current method. The class is illustrated in Figure 5.10.

<table>
<thead>
<tr>
<th>Method Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>+addMethod(in theMethod : DFA_Method)</td>
</tr>
<tr>
<td>+addMethod(in inClass : Class, in method : String, in paramTypes : Class[ ], in paramNames : String[ ], in onObject : DFA_Object)</td>
</tr>
<tr>
<td>+addConstructor(in inClass : Class, in paramTypes : Class[ ], in paramNames : String[ ], in onObject : DFA_Object)</td>
</tr>
<tr>
<td>+addStaticInitializer(in inClass : Class, in onObject : DFA_Object)</td>
</tr>
<tr>
<td>+startBlock()</td>
</tr>
<tr>
<td>+centerBlock(in associatedIdx : int)</td>
</tr>
<tr>
<td>+endBlock()</td>
</tr>
<tr>
<td>+addVariable(in varName : String)</td>
</tr>
<tr>
<td>+processAction(in varName : String, in action : int)</td>
</tr>
<tr>
<td>+processSuperAction(in varName : String, in action : int)</td>
</tr>
<tr>
<td>+processArrayAction(in varName : String, in action : int, in index : int)</td>
</tr>
</tbody>
</table>

Figure 5.10: Method Stack class

The next section examines how this structure can be used to locate *State Identifiers* associated with variables.

### 5.4 Searching in the Java Model

With the *Scoping Components* and *Scoping Component Containers* for Java defined, the next step in the model development is concerned with using these structure to locate variables. Information related to the actions performed upon variables in the program being tested needs to be routed through the model to the corresponding *DFA_Variable* object. As the memory location of the variable is not available within Java, the model must replicate the languages scoping rules.

#### 5.4.1 Processing Actions

The *processAction* methods located in the classes of the meta-model are used to implement the search for the appropriate *DFA_Variable*. Each of these methods performs a search on the associated structure and results in the action being passed to a *DFA_Variable* object, where it is used to trigger state transitions. The standard approach for this involves the *Scoping Component* searching its local collection of variables, and then delegating to another *Scoping Component* if required.
5.4.2 Local Variables

The Java language specification [14] indicates that variable identifiers are located by searching the local declarations first. A local variable will be accessed if the variable identifier is declared prior to the access within the current block, within any previous blocks, or as a method parameter.

The DFA_Method’s processAction method is used to start the search for the DFA_Variable. Since all of the method’s variables are contained within DFA_Blocks, the request is passed to the DFA_Method’s current DFA_Block. The DFA_Block checks its variables collection for a matching name. If the DFA_Block does not contain a matching variable, it passes the request to the previous block until a variable is located or the last block is searched. If the variable is not found within the method’s blocks, the DFA_Method’s associated DFA_Object is searched. The implementation for processAction of DFA_Block is shown in Listing 5.2, and DFA_Method’s implementation is in Listing 5.3.

5.4.3 Fields

If the search for the identifier fails to find a local variable, the identifier refers to a field.\(^4\) The field may exist either as a static field, instance field, or as a constant declared within an

\(^4\) Ambiguous cases do not need to be considered as they would be detected at compile time.
class DFA_Method
{
  private java.lang.Class declaredIn;

  public void processAction(String varName, int action)
  {
    if (!block.processAction(varName, action, this))
      currentObject.processAction(varName, action, this, declaredIn);
  }
}

Listing 5.3: Implementation of processAction in DFA_Method

Java provides a rich reflection API [30] that can be used to shortcut this search. An implementation of this search is shown in Listing 5.4. Initially, the class is queried for a declaration of the requested field; when this fails to find a match, the class’s interfaces are queried. If these queries fail to locate a field, the search is repeated for the class’s parent class. This will work its way up the inheritance hierarchy until the java.lang.Object class is encountered. The line if(from == null) break; occurs only where no field could be found. For the Java implementation, this will result in an error being reported. The Field object which is returned from this method can then be queried to determine which DFA_Variable should be used.

Instance Fields

For the DFA_Object class, the processAction method is used to inform the object that one of its instance variables may have been used. This method takes the variable’s name, the action type, the method in which the action was initiated, and the class with which the variable search starts. The getField method shown in Listing 5.4 is then used to locate the class in which the variable is declared. If this is an instance field, the DFA_Object uses its instanceVariables collection to locate the DFA_Variable object. This is done by creating a hash key from both the variable’s name and the declaring class’s name. When the variable is located, the action is passed to the DFA_Variable object. The implementation of DFA_Object’s processAction method is shown in Listing 5.5.

A quick examination of the code in Listing 5.5 appears to offer the opportunity to examine the modifiers of the Field in the processAction method. This would allow the code to determine if the field is a static or instance field, and then to directly pass the message to the corresponding DFA_Object. An example of this is shown in Listing 5.6. This implementation
import java.lang.reflect;

public class DFA_Object {
    ...

    private static Field getField(Class from, String varName) {
        do {
            try {
                return from.getDeclaredField(varName);
            } catch (NoSuchFieldException e) {
                Class[] interfaces = from.getInterfaces();
                for (int idx = 0; idx < interfaces.length; idx++) {
                    try {
                        return interfaces[idx].getDeclaredField(varName);
                    } catch (NoSuchFieldException ex) {
                    }
                }
                from = from.getSuperclass();
            }
        } while (from != null);
        throw new AssertionError("Model Failure");
    }

Listing 5.4: Code to locate a field in Java
public class DFA_Object
{
    ...

    public void processAction(String varName, int action,
                               DFA_Method where, Class startSearch)
    {
        Field theField = getField(startSearch, varName);
        doProcessAction(varName, action, where, theField.getDeclaringClass());
    }

    private void doProcessAction(String varName, int action,
                                  DFA_Method where, Class inClass)
    {
        DFA_Variable ia = null;

        if(instanceVariables != null)
        {
            // Processes all instance field accesses
            ia = (DFA_Variable)instanceVariables.get(inClass.getName() + varName);
        }

        if(ia != null)
        {
            // Variable is a field of this object, pass request to DFA_Variable
            ia.processAction(action, where);
        }
        else
        {
            // Variable is a static field of an interface or a parent class
            DFA_Object oid;

            // Locate the correct DFA_Object
            oid = DFA.getDFAObject(inClass);
            oid.doProcessAction(varName, action, where, inClass);
        }
    }
}

Listing 5.5: Implementation for processAction in DFA_Object
public void processAction(String varName, int action,
    DFA_Method where, Class startSearch)
{
    Field theField = getField(startSearch, varName);
    if (Modifier.isStatic(theField.getModifiers()))
    {
        DFA_Object oid;
        // Locate the correct DFA_Object
        oid = DFA.getDFAObject(inClass);
        oid.doProcessAction(varName, action, where, inClass);
    }
    else
    doProcessAction(varName, action, where, theField.getDeclaringClass());
}

Listing 5.6: Alternative implementation of DFA_Object’s processAction

will result in a StackOverflowException in the case where a field is defined during the loading
of a class. This is caused by the DFA_Class calling processAction to indicate the definition
of a static field during construction. As the field is static, the processAction method will call
getDFAObject, line 11. As the DFA_Class’s constructor is yet to complete, the object does
not exist within the Heap Space, and a new DFA_Class object is created, resulting in an
infinitely recursive call.

Static Fields and Interface Constants

The searching for static fields is also performed in the DFA_Object class. The else clause on
line 29 of Listing 5.5 performs the delegation from the object to the class in which the variable
is declared. The DFA_Class does not override the processAction method. The same can be
said for interface constants. In this case the Field’s declaring class refers to the interface,
which is also a DFA_Class object.

Object other than this

Fields may also be accessed on other objects, i.e. objects other than this. The searching method
described above only caters for access of the current object’s fields. To support the accessing
of fields on other objects, the Heap Space Scoping Component Container must provide the
ability to retrieve a DFA_Object via an object reference. In the cases where a field is accessed
on another object, a variant of the standard event can be used to indicate the object upon which
5  JAVA MODEL IMPLEMENTATION

Listing 5.7: Accessing another object’s fields

class MyClass
{
    public static final MyClass instance = new MyClass();
    public int value;
}

class TestClass
{
    void testMethod()
    {
        MyClass x = MyClass.instance;
        x.value = 100;
    }
}

the action has occurred. For example, the code in Listing 5.7 demonstrates the use of another object’s fields. MyClass contains a static field named instance, which is accessed via the MyClass class object, and an instance field named value that is accessed via the x reference variable. On line 11, the instance field of MyClass is referenced. This will be performed by searching the Heap Space for the corresponding DFA_Object. The same logic is applied for line 13, in which the value field is accessed on the x variable. In this case the Heap Space will be used to locate the DFA_Object related to the object referred to by x.

In addition to accessing public fields of other objects, the model must be able to handle an object that directly accesses fields of its super class. This is implemented in the processAction method of the DFA_Object. The startSearch parameter is used to indicate the class from which to begin the search.

5.4.4 Arrays

Within the meta-model DFA_Array instances are located in the same way as objects accessed via references. The DFA_Object that corresponds to the array is contained within the Heap Space. When an action is performed on an array, the model will search the Heap Space for the associated DFA_Object. When this object is located it can be cast to a DFA_Array, and the index of the array element that has been accessed can be passed in to the processAction method. The index can then be used to locate the appropriate DFA_Variable instance, to which the message is passed.
5.5 Summary

This chapter proposed a Java meta-model consisting of two main Scoping Components and two Scoping Component Containers. Scoping Components were defined to represent methods, which contain blocks, and objects, which include classes, interfaces, and arrays. Within the meta-model variables are represented using the DFA.Variable class. Instances of this class are responsible for performing dynamic data flow analysis for a single variable, and are located by search functionality implemented in the model.

When analysing a program the meta-model must be informed of a number of events within the running program. These include actions on variables, as well as operations that manipulate the meta-model such as method calls and returns, object creation and destruction, and class loading and unloading. These events must be communicated to the meta-model in order to perform the dynamic data flow analysis.
Part III

Collecting Action Information
Chapter 6

Program to Model Interactions

This chapter outlines the issues related to using the dynamic data flow model from within a program under analysis. It concludes with an examination of existing approaches for communicating between object oriented programs under analysis, and their data flow information model.

6.1 Program Instrumentation

The information for dynamic data flow analysis is maintained within a model as discussed in Part II. In order for this model to be used to perform dynamic data flow analysis, information must be gained from an executing program. Huang [17] proposed the use of source code probes for this purpose. Using this approach probes are inserted into the source code of the program prior to compilation. The resulting source code is then compiled and when executed the probes extract the required information.

While existing approaches have focused on source code instrumentation, several other approaches exist for gathering this information. These approaches range from compile time approaches to runtime approaches. Compile time approaches embed probes into the program to be executed, while runtime approaches modify the environment in which the program is executed. We presented a number of these approaches in [3], and will detail them further in chapter 7. The remainder of this chapter examines the existing approaches.
6.2 Instrumentation Issues

At its most basic level the approach used to extract information from the running program must be able to report the actions that are performed upon the program’s data elements. More specifically it must ensure that only actions that actually occurred are reported. While this may appear to be a trivial requirement, it is complicated by a number of features within different programming languages. These features include short circuit evaluation, exceptions, references or pointers, and object lifetime.

Short circuit evaluation refers to the ability for boolean conditions to only be partly evaluated in certain circumstances. For example, the condition \( c = a \land b \), will not evaluate \( b \) if \( a \) is \texttt{false}. This must be taken into consideration when reporting actions from the program under analysis.

Exceptions in languages such as Java allow for a structured approach to error handling. When an exception is thrown, control flow is directed to an appropriate \texttt{catch} block if one exists, otherwise the current thread is terminated. The Java source code in listing 6.1 illustrates a case where an \texttt{ArithmeticException} exception occurs. This exception occurs due to the divide by zero, and results in the value of \( d \) not being referenced. The approach used to collect information from the running program must ensure that changes to control flow due to exceptions do not cause information to be reported incorrectly.

References, or pointers, allow data elements to be referenced by multiple variables within the program. In order for the model to be able to correctly associate actions with data elements, the approach must be able to uniquely identify data elements accessed via references. In object
oriented languages such as Java and C#, all objects are accessed via references, and therefore this is of significant importance. When data elements can be accessed via references, their lifetime is no longer bound to the variable through which they are first referenced. The model needs to be informed when these data elements are deleted, and therefore the approach must be able to determine when a data element is deleted.

6.3 Existing Approaches

This section presents the two existing approaches for instrumenting object oriented programs. The approach used for C++ is presented first, followed by the approach used for Java.

6.3.1 C++ Probes

Chen and Low [9], indicate that their methodology uses the approach of Price [24] to instrument the source code, and identify the data items within the program. This approach uses source code probes, and identifies data elements by their memory location and size. In C++, the memory location of a variable is obtained using the address of operator &\texttt{()}, and the size of the variable can be obtained with the \texttt{sizeof} operator. By using the implicit data name for the data within the program, Chen and Low [9] are able to uniquely identify variables, including variables accessed via pointers.

The code in listing 6.2 illustrates the approach of Price [24]. With this approach the \texttt{ref} and \texttt{def} routines accept a pointer to the data element being accessed and then return that pointer after updating the model. As the probes are only executed when the action is performed this avoids the issues related to exceptions and short circuit evaluation.

```c
int main ()
{
    int a, b, c;
    ...
    ((int)*def(&a)) = ((int)*ref(&b)) + ((int)*ref(&c));
}
```

Listing 6.2: Source code probes in C++

With Chen and Low’s methodology, the probes within the program contain three values. The first value is the memory location of the variable under analysis. This information is used to identify the variable on which the action was performed. The second, indicates the type of action that has taken place; being one of either \texttt{defined}, \texttt{referenced}, or \texttt{undefined}. The final
value indicates the class of origin, which is the class in which the action was performed. When the action occurs within a global function, the class of origin is noted as ‘Global’.

**Evaluating the C++ Approach**

The methodology presented directly addresses some of the requirements presented in section 2.4. The following list describes the level to which each of the requirements is met.

1. The methodology indicates how variables can be tracked.
2. The scope, or type of variable is addressed by the methodology.
3. Targeted analysis is not addressed by the methodology.
4. The methodology does not indicate how errors are to be reported.
5. The methodology can be automated.

The methodology describes how actions can be tracked for variables via their memory location. This addresses any scope, and visibility related issues, requirements 1, and 2. Additionally, Chen and Low [10] provide a way for testing both variables and pointers. This ensures that all types of variables can be tested, further addressing requirement 2. What is not addressed is targeted analysis, requirement 3, and no details are provided on how the output allows a programmer to locate anomalies, requirement 4. An implementation of this approach can easily provide sufficient output to locate anomalies in the program’s source code, in fact the approach presented in [24] gives a good indication of how this requirement can be met. Targeted analysis, on the other hand, is not sufficiently addressed, and there is no obvious way that this requirement can be met.

To illustrate the issues related to targeted analysis of C++ code, let us reconsider the Point code from Listing A.1. An excerpt from this sample, showing `main`, is provided for convenience in listing 6.3. The `time` function on line 8, exists within the standard libraries. These libraries are not instrumented for dynamic data flow analysis, and as such the definition of the memory location referred to by `tnow` is not reported. It is also noted in the documentation of the `time` method, that this function does not set the value when an error occurs. Without knowing the semantics of the time function it is not possible to know which type of action should be applied to the value in `tnow`. If we indicate that the `tnow` variable is defined this will
be incorrect when an error occurs, and visa versa. Similar issues are present with other library functions, for example `scanf` and `fscanf`, and are also likely to occur with other libraries.

```cpp
#include <stdlib.h>
#include <time.h>
#include <iostream.h>

int main()
{
    time_t tnow;
    time(&tnow);
    srand(tnow);
    MovablePoint p(10,100,10);
    p.move();
    cout << p;
}
```

Listing 6.3: Main from C++ Point source code.

This methodology is capable of handling exceptions, and short circuit evaluation, but has some issues related to targeted analysis that require further investigation.

### 6.3.2 Java Probes

Boujarwah et.al [2] use twelve cases to indicate where probes must be inserted into Java source code. Each of the cases addresses a specific issue, and indicates the location at which probes are to be inserted. The contents of the probe is determined by the type of variable being used, as detailed in section 3.2.1. The probes used in [2] write their details to `System.out` as a single line which, in all cases, contains the identifiers from the following list. In this section we will illustrate this approach using the example from Appendix A.2. To remain consistent with their approach we will also use the `System.out` and the identifiers from the following list. For example, the definition of the `x` instance field on line 3, is instrumented using `System.out.println("(x, Instance, main,(String[]), ParticleTest, p, -, -, -, Particle, d)");`. The items marked with a ‘–’, indicate that no value is supplied for this identifier. In this case, there is no value supplied for the method name used for return values, its arguments, or class. The identifiers for the data flow actions include:

1. The variable name upon which the action is being performed.
2. The type of variable, being either Class, Instance, or Local.
3. The name of the method, for identification of the associated object or local variable.
6. PROGRAM TO MODEL INTERACTIONS

4. The arguments of the above method.

5. The name of the class containing the above method.

6. The object name.

7. The method name for actions on return values.

8. The parameter types for this method.

9. The class containing this method.

10. The name of the class containing the variable, in the case of instance and static fields.

11. An identifier for the action performed: d for defined, u for undefined, and r for referenced.

The first case addresses the probes required to indicate the actions performed at the start of program execution. This case indicates that definition actions for all class fields and interface constants must be performed as the first action within the first executed method. For our example, the first method is the main method on line 38 from Appendix A.2. Within this method we need to insert probes to report the definition of the static variable rng. The code for this is shown in Listing 6.4.

Listing 6.4: Probes at program startup

```java
class Particle {
    private static final java.util.Random rng = new java.util.Random();
}

class ParticleTest {
    public static void main(String[] args) {
        System.out.println("(rng.class,-,-,-,-,-,-,-,Particle,d)");
        ...
    }
}
```

Case two describes the probes required at the start of a method call. The parameters with a primitive type require definition probes to be inserted before the first action within the method.
\begin{verbatim}

class Particle
{
    public Particle(int x, int y)
    {
        System.out.println("(x, Local, Particle, (int, int), Particle, -, -, -, -, d)"");
        System.out.println("(y, Local, Particle, (int, int), Particle, -, -, -, -, d)"");
        ...
    }
}
\end{verbatim}

Listing 6.5: Probes for the start of methods

\begin{verbatim}

class Particle
{
    ...

    public Particle(int x, int y)
    {
        this.x = x;
        System.out.println("(x, Local, Particle, (int, int), Particle, -, -, -, -, r)"");
        System.out.println("(x, Instance," + determineObject.methodName + "+" + determineObject.methodArgs + "+" + determineObject.className + "+" + determineObject.objectName + "+" + Particle.d"");
        ...
    }
}
\end{verbatim}

Listing 6.6: Probes within methods

In our example the \textit{Particle} constructor accepts two primitive parameters of type \texttt{int}. The result of instrumenting this method is shown in Listing 6.5.

The locations for the probes within the bodies of methods is described in case three. This case indicates that probes for reference and definition actions are inserted on the line \texttt{after} the statement in which the actions are performed. Listing 6.6 illustrates the probes required for the body of the \textit{Particle} constructor. In this illustration, the probe for the definition of the \texttt{x} instance field makes use of the \texttt{determineObject} class to report the details of the current object. The values in the \texttt{determineObject} class are set by the calling method prior to making the call.

Probes for the actions that occur when a method terminates are considered in cases four and nine. Specifically, case four describes method return values, while the un-define actions for
class Particle
{
    public Particle(int x, int y)
    {
        ...
        System.out.println("x, Local, Particle, (int, int), Particle, -, -, -, -, u");
        System.out.println("y, Local, Particle, (int, int), Particle, -, -, -, -, u");
    }

    public int getX()
    {
        ...
        System.out.println("-Instance, " + determineObject.methodName +
        ", " + determineObject.methodArgs +
        ", " + determineObject.className +
        ", " + determineObject.objectName +
        ", .getX(), Particle, Particle, d");
        return x;
    }
}

Listing 6.7: Probes at the end of methods

local variables are presented in case nine. In both of these cases the probe for these actions must be performed prior to the return statement, as this statement will terminate the current method. When a method returns data, probes for the reference of the variables returned, and probes for the definition of the returned value must be inserted. Where a constant value is returned the reference action is omitted. Additionally, probes must also be inserted to indicate the un-defining of the method’s local variables. Listing 6.7 illustrates this for the example presented. At the end of the constructor, un-define actions are instrumented for the local variables x and y. The getX method returns an int value, and probes must be instrumented to report the definition of this value. As this is an instance method, the object identification information is obtained from the determineObject class.

Case five deals with object construction. When an object is created, probes must be inserted to define all explicitly initialized instance fields, default initialization is ignored. Instance fields that are not initialized do not require any probes. In the example shown in Listing 6.8, the definition of p’s x instance field is reported. Fields y and rng do not get defined, as y is not initialized in the class, and rng is a static field.

The probes required to handle the value returned from a called method are provided in case six. The return value is defined within the method, and so case six indicates that probes need
class Particle
{
    private int x = 0;
    private int y;
    private static final java.util.Random rng = new java.util.Random();
    ...
}

class ParticleTest
{
    public static void main(String[] args)
    {
        System.out.println("(x, Instance, main, (String[]), ParticleTest, p, -, -, -, Particle, d)");
        Particle p = new Particle(100, 100);
        ...
    }
}

Listing 6.8: Object construction probes

class ParticleTest
{
    public static void main(String[] args)
    {
        ...
        Particle p1 = p;
        int p1x = p1.getX();
        System.out.println("(-, Instance, main, (String[]), ParticleTest, p1, getX, (), Particle, Particle, r)");
        System.out.println("(-, Instance, main, (String[]), ParticleTest, p1, getX, (), Particle, Particle, u)");
    }
}

Listing 6.9: Probes referencing return values

to be inserted to report the reference and un-define of this value within the calling method. Methods that return no data, i.e. void methods, do not need to be considered. These probes must be inserted after the call to the method, and are illustrated in Listing 6.9. This shows the probes for the value returned by the getX method.

Case seven deals with the circumstance where an object is instantiated as an argument to a method, for example, System.out.println(new Particle()). In these cases, probes are inserted before the source code line that contains the object construction. These probes indicate the definition actions performed by the construction of the object. More probes are added on the
class ParticleTest
{
    public static void main(String[] args)
    {
        ...
        System.out.println("(x, Instance, main, (String[]), ParticleTest, p, -, -, -, Particle, u)");
        System.out.println("(y, Instance, main, (String[]), ParticleTest, p, -, -, -, Particle, u)");
    }
}

Listing 6.10: Object destruction probes

line following the call for the destruction of the object. In this case a probe would be inserted prior to the line, indicating the definition of the x instance field of the object constructed, and two probes would be inserted on the following line indicating the un-defining of variables x and y. The object name for these instance is taken from the method’s parameter name. For the example given the object name is x as println is defined as public void println(String x) in the Java API [30].

Object destruction is dealt with in cases eight and nine. Case nine, as we have discussed earlier, deals with the destruction of variables at the end of a method. The illustration of this case indicates that probes should be inserted for the destruction of objects that are referred to via local variables, excluding those referred to via the method’s parameters. In our case, this requires probes to be inserted at the end of the main method as shown in Listing 6.10. Probes for the un-defining of rng are not instrumented as this is a static field. Case eight indicates that probes to report the destruction of the object should be inserted after a call to the object’s finalize method.

Program termination is addressed in case ten. Probes to indicate the termination of the program and the un-defining of all static fields are inserted as the last statement within the first executed method. These probes are inserted at the end of the main method for the example presented, see Listing 6.11.

Case eleven deals with the probes for reference variables. To enable tracking of the object names, a probe must be inserted before each reference variable assignment. This must be applied for reference variable assignment, parameter passing, and returned values. The examples

---

1There is an issue with this as objects are not necessarily destroyed in this circumstance. Discussion of this is left for later in the chapter.
presented in [2] indicate that these probes must be inserted before the assignment, in case of variable assignment and parameter passing, and after for assignment from return values. In Listing 6.12, a probe has been added within the `main` method to indicate that `p1` refers to the same object as `p`.

The `determineObject` class as defined in [2], shown in listing 6.13, is used to pass the object name for the current object to a called method. The setting of the values in the `determineObject` class are shown in listing 6.13. This information is used in the called method to identify the object referred to by this. Several of the previous examples demonstrate the use of this information.

When the instrumented program is executed the data flow information is written to the console window. Each line of this output refers to an action performed upon a variable. Actions recorded via object names must be grouped together based on the probes reported for the assignment of reference variables. This ensure that actions performed on instance variables are correctly associated with their objects during analysis.
class determineObject
{
    public static String objectName;
    public static String methodName;
    public static String methodArgs;
    public static String className;
}

class ParticleTest
{
    public static void main(String[] args)
    {
        ...
        determineObject.objectName = "p1";
        determineObject.methodName = "main";
        determineObject.methodArgs = "(String[])";
        determineObject.className = "ParticleTest";
        int p1x = p1.getX();
    }
}

Listing 6.13: Identifying the current object

Java Approach Evaluation

The approach presented by Boujarwah et al. [2], meets only some of the requirements for dynamic data flow analysis presented in section 2.4. There are several key flaws with this approach, with the main issues being the tracking of instance variables via their object name, and only providing data flow analysis for primitive types. The details of how this approach meets the requirements described in section 2.4 are presented in the following list.

1. The way in which objects are tracked makes it difficult, and in some cases impossible, to track instance variables reliably.

2. With this approach incorrect or incomplete actions may be reported for variables.

3. The approach does not provide a mechanisms to allow the analysis of reference variables.

4. Targeted analysis is not possible with this approach.

5. The output from this approach provides some information for tracking the location of an anomaly.

6. The approach is automated.
The Java language specification lists the primitive types for Java as being integral types (byte, short, int, long, and char), floating point types (float and double), and the boolean type. Restricting data flow analysis to only these types ignores a large majority of the data flow actions that occur within a Java program. For example, no data flow actions are reported by the execution of `String s = new String("Hello World").` As we do not know the internal implementation of the `String` class we cannot report any actions on associated primitive types, and the define action performed on `s` is not recorded as `String` is a reference type.

When using data flow analysis to test a program, the actions within our program’s source code are of far more relevance than the actions that occur entirely within external libraries. Using the above example, `String s = new String("Hello World")`, the definition of `s` is relevant to our current analysis, while the internal actions that define the char array elements within the `String` class are of little interest. In addition, if the definition of these variables are reported, all other actions related to these variables must also be reported. Without this, anomalies may be incorrectly reported for these variables. Therefore, to ensure correct analysis, the internal structure of all classes used must be known, and all of these classes must be instrumented to report actions on their primitive instance variables. This requirement cannot be met for Java applications without re-writing parts of the standard library.

The object identification information for the current object is contained within the `determineObject` class. This information is set in the calling method as shown in Figure 6.13. To report the data flow actions within a method, the previous method must be instrumented to provide the information for the `determineObject` class. This causes a number of issues when targeted analysis is considered. Figure 6.1 provides a UML sequence diagram that illustrates one such issue with this approach. This diagram depicts the use of the `determineObject` class by three related classes; Class1, Class2, and Class3. Class1 and Class3 are instrumented for data flow analysis, while Class2 is not. Prior to the call to Object2, Object1 sets the information on the `determineObject` class. As Class2 is not instrumented it does not make use of this information, nor does it change the information prior to its call to Object3. Object3 is instrumented for analysis, and reads what it believes to be its identification information from the `determineObject` class. As this was not changed by Class2, all actions reported for the instance variables of the current object will have the incorrect object name, ‘Object2’.

Another targeted analysis issue, relates to the values returned from methods. Using the sequence diagram from Figure 6.1, if the method call to Object3 returns a value, probes will
be inserted to indicate that this value has been defined. As Object2 is not instrumented, this value will never be referenced. In addition to this, if Object2’s method returns a value, probes will be inserted to report the reference and undefine actions on this value in Object1. This will result in an undefine-reference anomaly being reported by the analysis, as Object2 will not have reported the definition of this value. Polymorphism [4] further complicates this issues as one method call may be made on various objects, and therefore call instrumented code in one instance and uninstrumented code in another.

Placing the probes for a statement on the following line may result in incorrect or incomplete reporting of actions. One Java statement may perform a number of actions on variables. Some of these actions may result in exceptions being raised, causing the remainder of the statement to remain unexecuted. For example, Listing 6.14 illustrates some Java code that contains a line that will raise an exception. The execution of foo.bar(a, b/c, d) will raise the exception ‘java.lang.ArithmeticException: / by zero’. This exception will bypass the reporting of the reference actions that follow on the next line. If we place the reporting of the reference actions prior to the statement, then the reference of variable d is incorrectly reported as the parameters are evaluated from left to right, and b / c causes the exception.

In addition to these errors, there are several Java related issues with [2] that result in incorrect reporting of actions. The following list enumerates these issues.

1. The approach does not handle class loading correctly. According to case one, all initial-
6.14: Instrumentation with exceptions

Manipulated static fields are defined as the first action within the first method. However, the Java language specification [14] indicates that a class is initialized on first use. Using case 1 will result in incorrectly reporting the definition of static fields from classes that are not used in a given execution.

2. Case eight indicates that an object is destroyed when its finalize method is called. The finalize method does not destroy an object, rather it is called by the garbage collected when the object is reclaimed [14].

3. Probes for the destruction of an object that is constructed locally, are instrumented at the end of the method; cases seven and nine. This may result in the incorrect reporting of the un-define action for the fields of the object. An object is only reclaimed by the garbage collector once it is no longer referenced. For example, given the code in Listing 6.15, probes for the un-defining of the fields of object f are inserted at the end of the AddNew method. However, the object referred to by f is still referenced within the ArrayList.

4. Case ten implies that the program terminates at the end of the first called method. This is not always the case. Some example where this is not the case include; Applets, where the first method returns before program termination, analysis of class libraries where the lifetime of the application is handled externally, and applications with multiple threads. Reporting the termination of the program at the end of the first method may result in incorrect analysis in these cases.
5. Unloading of classes is incorrectly handled by this approach. According to the Java language specification [14], a class may be unloaded when it is no longer needed. Case ten however, reports the un-defining of static fields at the end of the first method. This will cause similar problems to those discussed in point 1 above, and also incorrectly reports actions for classes that were never loaded.

In general, the approach in [2] fails to perform sufficiently to be of practical use. Actions are incorrectly reported in numerous cases, analysis of reference variables is ignored, and targeted analysis is not possible. Several of the foundations for this approach must be reconsidered if a practical approach to dynamic data flow analysis of Java programs is to be provided. As the concurrent data flow analysis approach in [28] uses this approach, it also suffers from these same problems.

6.4 Summary

Data flow analysis models require information from running programs regarding the use of the program’s data elements. Existing approaches have used source code probes to report this information. Chen and Low [9] use the approach of Price [24] to perform source code instrumentation for C++. This approach is effective in avoiding issues related to short circuit evaluation and exceptions. Java, however, does not provide sufficient facilities to enable this same approach to be used and therefore Boujarwah et al. [2] propose a new approach. Unfortunately their approach suffers from a number of issues. The following chapter will present a number of new approaches for Java.
Chapter 7

Program to Java Model
Implementation

Chapter 4 introduced a meta-model based approach for performing dynamic data flow analysis. Chapter 5 proposed a meta-model for Java and outlined its functionality. Chapter 5 also outlined a number of events that must be communicated to the model for it to function. This chapter describes several approaches for communication these events from the program under analysis to the meta-model.

In previous work we proposed a number of techniques for collecting dynamic data flow analysis information from Java programs [3]. This included (i) source code instrumentation, (ii) byte code instrumentation, and (iii) the Java Platform debugging architecture [32]. Each of these approaches are illustrated in more detail in this chapter.

Each approach describes how it is able to communicate the events required by the meta-model as detailed in Section 4.2.4. For Java these events include the following:

1. Creation of new, and destruction of existing, Scoping Components including:
   
   (a) Loading and unloading of Classes.
   (b) Creation and destruction of objects.
   (c) Start and end of methods.
   (d) Start and end of blocks.
   (e) End of methods and blocks via Exceptions.

2. Declaration of local variables.
3. Messages for all actions described in Section 5.1 Variable Analysis for Java.

Each of the proposed approaches will also describe how it meets the requirements expressed in the previous chapters (Chapter 6 in section 4.3.2), namely:

1. A technique for identifying objects must be presented.

2. The approach must address targeted analysis.

3. The implementation must ensure that only events that occur are communicated to the meta-model.

4. All events for the variables within the targeted scope must be communicated to the meta-model.

5. The communication method used must not change the semantics of the program, i.e. the tested program is observably equivalent [7] to the original.

### 7.1 Interface to the Model

Regardless of the approach used, the instrumented program needs to be able to interact with the model. This section examines this interface. For this implementation, a DFA class has been added to the meta-model for this purpose. All of the methods within this class are declared as static to enable easy access regardless of the approach used.

The DFA class provides methods to indicate the following actions to the meta-model, shown in Figure 7.1:

1. Actions performed upon variables, including:
   
   (a) Define actions (local, array, and public field), define
   (b) Define actions on a super class field, superDefine
   (c) Reference actions (local, array, and public field), reference
   (d) Reference actions on a super class field, superReference

2. Creation of local variables, addLocalVariable

3. Start of a method, including:
7 PROGRAM TO JAVA MODEL IMPLEMENTATION

### Figure 7.1: The DFA class

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objectMethodStarted</code></td>
</tr>
<tr>
<td><code>classMethodStarted</code></td>
</tr>
<tr>
<td><code>staticInitializerStarted</code></td>
</tr>
<tr>
<td><code>constructorStarted</code></td>
</tr>
<tr>
<td><code>methodEnded</code></td>
</tr>
<tr>
<td><code>blockStarted</code></td>
</tr>
<tr>
<td><code>blockEnded</code></td>
</tr>
<tr>
<td><code>catchFinallyStarted</code></td>
</tr>
<tr>
<td><code>shutdown</code></td>
</tr>
</tbody>
</table>

(a) Instance methods, **objectMethodStarted**

(b) Static methods, **classMethodStarted**

(c) Static initializers, **staticInitializerStarted**

(d) Constructors, **constructorStarted**

4. End of a method, **methodEnded**

5. Start of a new block, **blockStarted**

6. End of a block, **blockEnded**

7. Exception handling, including final blocks, **catchFinallyStarted**

8. Disposing of the remaining model object upon program termination: **shutdown** method which is called via a shutdown hook attached to the Java Runtime.

#### 7.1.1 Managing Methods

The start of each method must be instrumented with a call to one of either **objectMethodStarted**, **classMethodStarted**, **staticInitializerStarted**, or **constructorStarted**, and must
conclude with a call to `methodEnded`. Listing 7.1 illustrates the use of these methods. Each of these methods supplies sufficient information for the meta-model to be able to construct a matching `DFA_Method`, or subclass thereof.

The `staticInitStarted` method is used to indicate the start of a static initializer. Listing 7.1 has a static initializer for the `MethodStartSample` class at line 5. There is only one version of the `staticInitStarted` method, which accepts one parameter indicating the class in which the initializer is declared. The class can be retrieved using the `.class` option on the class’s name, as shown in the listing. Internally the `DFA` class informs the `MethodStack` to add a new `DFA_StaticInitializer`.

The start of a static method is indicated by a call to `classMethodStarted`, as shown on line 17 of Listing 7.1. The `classMethodStarted` method requires several pieces of information from the program. These include the class in which the method is declared, the name of the method, and the details of the method’s parameters via an array of classes and a parallel array of parameter names. To simplify this, two versions of the method are supplied. One version omits the details of the parameters to allow easy instrumentation of methods without parameters. The implementation of these two methods are shown in Listing 7.2.

To indicate the start of an instance method, the `DFA`’s `objectMethodStarted` method is called. In a similar fashion to the `DFA`’s `classMethodStarted`, there are two version of this method. One that requires parameter information, and one that does not. `DFA`’s `objectMethodStarted` accepts three parameters in addition to the two parameters used for parameter information. The three parameters accept the object reference, the `Class` in which the object is defined, and the method’s name. The object reference is used to request the `DFA_Object` instance from the heap space. The implementation of these methods is shown in Listing 7.3.

The `DFA`’s `constructorStarted` method is used to indicate that a constructor is to be added to the method stack. As the constructor may accept a number of parameters, two methods will exist with `DFA` to handle this. In addition to the parameter information, the `DFA`’s `constructorStarted` method accepts two parameters, the object reference, and the class in which the constructor is declared. The implementation for `constructorStarted` is shown in Listing 7.4.

The constructor is also used to indicate the creation of a new object. When a constructor starts, the `DFA` class determines if this object has been identified\(^1\) before, and if not creates a new `DFA_Object` to represent the object within the meta-model. This is only done within the

\(^1\)This will occur when the object inherits multiple instrumented constructors.
import swin.ddfa.DFA;

public class MethodStartSample
{
    static
    {
        DFA.staticInitStarted(MethodStartSample.class);
        try {}
        finally
        {
            DFA.methodEnded();
        }
    }

    public static void main(String[] args)
    {
        DFA.classMethodStarted(MethodStartSample.class, "main",
            new Class[] { String[].class }, new String[] { "args" });
        try {}
        finally
        {
            DFA.methodEnded();
        }
    }

    public MethodStartSample()
    {
        DFA.constructorStarted(this, MethodStartSample.class);
        try {}
        finally
        {
            DFA.methodEnded();
        }
    }

    public void Test(Object o)
    {
        DFA.objectMethodStarted(o, MethodStartSample.class, "Test",
            new Class[] { Object.class }, new String[] { "o" });
        try {}
        finally
        {
            DFA.methodEnded();
        }
    }
}

Listing 7.1: Method Start Instrumentation
class DFA
{
    ...
    private static void methodStarted(Class from, String name,
            Class[] param, String[] paramNames, DFA_Object oid)
    {
        getMethodStack().addMethod(from, name, param, paramNames, oid);
    }
    
    public static void classMethodStarted(Class from, String name,
            Class[] param, String[] paramNames)
    {
        DFA_Object oid = allObjects.getDFAObjectFor(from);
        methodStarted(from, name, param, paramNames, oid);
    }
    
    public static void classMethodStarted(Class from, String name)
    {
        classMethodStarted(from, name, NO_PARAM, NO_PARAMNAMES);
    }
}

Listing 7.2: Class Method Started

class DFA
{
    ...
    public static void objectMethodStarted(Object o, Class from, String name,
            Class[] param, String[] paramNames)
    {
        DFA_Object oid = allObjects.getDFAObjectFor(from);
        methodStarted(from, name, param, paramNames, oid);
    }
    
    public static void objectMethodStarted(Object o, Class from, String name)
    {
        objectMethodStarted(o, from, name, NO_PARAM, NO_PARAMNAMES);
    }
}

Listing 7.3: Object Method Started
class DFA
{
...  
public static void constructorStarted(Object o, Class from,
    Class[] param, String[] paramNames)
{
    DFA_Object oid = allObjects.getDFAObjectFor(o);
    //Creates the DFA_Object if it does not already exist
    if (oid.isUnknown())
    {
        oid = new DFA_Object(o);
        allObjects.add(oid, o);
    }
    getMethodStack().addConstructor(from, param, paramNames, oid);
}
public static void constructorStarted(Object o, Class from)
{
    constructorStarted(o, from, NO_PARAM, NO_PARAMNAMES);
}

Listing 7.4: Constructor Started

constructor to ensure that only objects within targeted code are loaded in the meta-model.

7.1.2 Instrumenting Variable Access

There are three ways in which variables can be accessed; directly via the variable’s name, on
another object via the object’s\(^2\) reference, and within the parent class via the super keyword.
Variables may also be accessed for definition or reference. Source code probes need to be
inserted into the program’s source code to indicate these actions. The probes should be inserted
prior to the line of code in which the action occurs. Listing 7.5 illustrates the instrumentation
required for the different ways in which variables can be accessed.

The DFA class contains two methods to support the use of variables directly via their name;
methods reference and define. Both of these methods call the DFA’s processAction method,
passing in an identifier for the type of action performed. The processAction method passes
the action to the model’s MethodStack, which then delegates it to the top method. The imple-
mentation of these methods is shown in Listing 7.6.

When the code accesses a variable on an object other than the current object, the model
\(^2\)Including via array references.
```java
import swin.ddfa.*;

public class AllVariables
{
    static int statInt;
    int instInt;

    public AllVariables()
    {
        DFA.constructorStarted(this, AllVariables.class);
        try
        {
        }
        finally
        {
            DFA.methodEnded();
        }
    }

    public static void main(String[] args)
    {
        DFA.classMethodStarted(AllVariables.class, "main",
                               new Class[] { String[].class }, new String[] { "args" });
        try
        {
            DFA.reference("args");
            if(args != null && args.length > 0)
            {
                DFA.referenceArrayElement(args, 0);
                System.out.println(args[0]);
            }
            DFA.addField("av");
            DFA.define("av");
            AllVariables av = new AllVariables();
            DFA.reference("av");
            DFA.objectDefine("instInt", av, AllVariables.class);
            av.instInt = 10;
            DFA.reference("av");
            DFA.objectDefine("statInt", av, AllVariables.class);
            av.statInt = 20;
            DFA.reference("statInt");
            System.out.println(statInt);
        }
        finally
        {
            DFA.methodEnded();
        }
    }
}
```

Listing 7.5: Instrumenting for Variables
### 7.6 Define and Reference

needs to be informed of the object which was accessed. The DFA class contains two methods to handle these situations, objectDefine and objectReference. These methods take the name of the variable as well as the reference to the object upon which the variable was accessed. See Listing 7.7 for the implementation.

The DFA class also provides methods to indicate array access, as shown in Listing 7.8. Within the meta-model, arrays are treated as object that are accessed using an array index. The DFA class provides two methods to interact with arrays, defineArrayElement and referenceArrayElement. Both of these methods accept the array as an object, and the index of the array element accessed as an integer. Internally these methods locate the DFA_Array object using getDFAObject, and the MethodStack is called to process the action.

Finally, variables accessed via the super keyword need to be addressed. For this, the DFA class provides the superDefine and superReference methods. The super keyword can only be used in the local context, and therefore, the methods accept only the name of the variable being accessed. The implementation for these methods is shown in Listing 7.9.
Listing 7.7: Object Define and Reference

class DFA
{
    ...

    private static void objectAction(String varName, Object o, Class c, int action)
    {
        DFA_Object oid = getDFAObject(o);
        if (false == oid.isUnknown())
            oid.processAction(varName, action, getMethodStack().getCurrentMethod(), c);
    }

    public static void objectDefine(String varName, Object o, Class c)
    {
        objectAction(varName, o, c, DEFINE);
    }

    public static boolean objectReference(String varName, Object o, Class c)
    {
        objectAction(varName, o, c, REFERENCE);
        return true;
    }
}

Listing 7.8: Array Define and Reference

class DFA
{
    ...

    public static void referenceArrayElement(Object o, int index)
    {
        DFA_Array oid;
        oid = (DFA_Array) getDFAObject(o);
        getMethodStack().processArrayAction(oid, index, REFERENCE);
    }

    public static void defineArrayElement(Object o, int index)
    {
        DFA_Array oid;
        oid = (DFA_Array) getDFAObject(o);
        getMethodStack().processArrayAction(oid, index, DEFINE);
    }
}
```java
class DFA {
    ...

    private static void superAction(String varName, int action) {
        getMethodStack().processSuperAction(varName, action);
    }

    public static void superDefine(String varName) {
        superAction(varName, DEFINE);
    }

    public static void superReference(String varName) {
        superAction(varName, REFERENCE);
    }
}
```

Listing 7.9: Super Define and Reference
7.1.3 Handling Blocks

The instrumented program must be able to communicate the start and end of blocks. The DFA class provides three methods to manage this, namely startBlock, endBlock, and catchFinallyBlock. The startBlock method indicates the start of a new block in the program under analysis. This method accepts one parameter that indicates the block’s identifier, which is used in reporting and managing exceptions. The endBlock method indicates the end of a block, this does not need to supply the block identifier as this always ends the current block. The catchFinallyBlock indicates that a catch or finally block has been entered. This method take one parameter that indicates the block identifier for the associated try block. This identifier is used to end blocks within the current method, and synchronize the meta-model with the current execution location. The addCatchVariable adds a new local variable to the method, and then performs a define action for that variable. These methods are illustrated in Listing 7.10.

```java
class ExceptionTest {
    public static void main(String args[]) {
        DFA.classMethodStarted(ExceptionTest.class, "main", new Class[]{args.getClass()}, new String[]{"args"});

        try {
            DFA.addLocalVariable("y");
            DFA.define("y");
            int y = 0;

            try {
                DFA.startBlock(1);

                DFA.addLocalVariable("x");
                DFA.define("x");
                int x = 10;

                try {
                    DFA.startBlock(2);
                    DFA.addLocalVariable("o");
                    Object o = null;
                    DFA.reference("o");
                    o.toString();
                    DFA.reference("x");
                }
            }
        }
    }
}
```

7 PROGRAM TO JAVA MODEL IMPLEMENTATION

```
7.10: Blocks and Exceptions

Listing 7.10: Blocks and Exceptions

7.2 Source Code Instrumentation

Existing approaches for dynamic data flow analysis have used source code instrumentation. Using this technique, the source code of the program under analysis is transformed to include a number of source code probes. These probes, when executed, perform the dynamic data flow analysis. This section presents an approach for using source code instrumentation to communicate events to the meta-model.

7.2.1 Identifying Objects

The identification of objects is required to enable the object to be retrieved from the Heap Space Scoping Component Container. Using source code instrumentation, this can be achieved by using the object’s `hashCode` method. The Java API documentation [30] lists `hashCode` as follows:

```
public int hashCode()
```
Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable.

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

- If two objects are equal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java programming language.)

This gives a good indication that in most cases the object's hashCode will provide sufficiently for identification purposes. However, the fact that non-equal objects may return non-distinct values needs to be further examined further. The equals method of the Object class is defined in the Java API [30] as:

```
public boolean equals(Object obj)
```

Indicates whether some other object is ”equal to” this one.

The equals method implements an equivalence relation on non-null object references:
• It is reflexive: for any non-null reference value x, x.equals(x) should return true.

• It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.

• It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

• It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

• For any non-null reference value x, x.equals(null) should return false.

The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

This provides a solution to one problem, but introduces another problem. The Heap Space can be implemented using Java’s Hashtable class. The DFA_Object instances can be stored within the Hashtable using the key determined from the object’s hashCode method. In order to ensure that this works correctly, the object reference itself could be used as the key. Unfortunately this will not work as ‘it is generally necessary to override the hashCode method’ [30]. Fortunately the System class contains a identityHash method that returns the value from the Object class’s implementation of hashCode.

Using the identityHash method means that the object itself cannot be used as the key for the DFA_Object. To overcome this issue a new class can be created. The new class will provide the hashCode and equals methods, ensuring that the correct DFA_Object is returned from the Hashtable. The code for this is shown in Listing 7.11.

The implementation shown for the ObjectIdentifier contains a reference to the object itself. Doing this will result in the object remaining reachable, and thereby making it impossible
Listing 7.11: Object Identification via hashCode

for the garbage collector to reclaim it. This can be avoided if the ObjectIdentifier uses a java.lang.ref.WeakReference to refer to the object. This is similar to using the java.util.-WeakHashMap, however in this case the object is weakly referenced rather than the key. An implementation of the new ObjectIdentifier class is presented in Listing 7.12. The hash for the object is calculated at construction as objRef.get() will return null once the object has been marked for collection by the garbage collector.

This implementation of the ObjectIdentifier still suffers from one shortcoming. The WeakReference’s referent is set to null once the object is marked for collection. In the case where the object has a finalize method it may be possible for the ObjectIdentifier to identify an incorrect object. This would require two objects to have the same identityHash value, and both to be available for collection by the garbage collector at the same time. In this case line 27 of Listing 7.12 will return true for either of the objects, as their hash values will be the same and the WeakReference’s get method will return null in both instances. This implementation will also have issues in the case where the object is made reachable in its finalize method. In these cases the WeakReference will refer to null even though the object is reachable again. This is the strongest form of object identification that is available without implementing a custom solution. The error cases are highly unlikely to occur. In the first case, the hash code method attempts to ‘return distinct integers for distinct objects’ [30]. Secondly, it is not common practice
import java.lang.ref.WeakReference;

public class ObjectIdentifier
{
    private WeakReference objRef;
    private int hash;

    public ObjectIdentifier(Object toIdentify)
    {
        objRef = new WeakReference(toIdentify);
        hash = System.identityHashCode(toIdentify);
    }

    public int hashCode()
    {
        return hash;
    }

    public boolean equals(Object o)
    {
        if(o instanceof WeakReference)
            return o == objRef;
        if(o instanceof ObjectIdentifier)
        {
            ObjectIdentifier oid = ((ObjectIdentifier)o);
            return oid.hash == hash && oid.objRef.get() == objRef.get();
        }
        Object obj = objRef.get();
        return obj == o;
    }
}

Listing 7.12: Object Identification with weak references
to ‘resurrect’ objects in their finalize method, and in these cases an alternative custom solution can be created.

### 7.2.2 Classes and Objects

In order to ensure that classes and objects are tracked successfully the meta-model must be informed of their creation. To ensure that a **DFA.Class** is loaded when it is accessed, a static initializer must exist within the class. When no static initializer exists in an instrumented class, one must be added. In these cases the static initializer should appear as shown in Listing 7.13.

To ensure that a **DFA.Object** instance is created for each object of a class, the class must have an instrumented constructor. In the case where there is no constructor, one must be added. If a constructor is added it must be a public default constructor as shown in Listing 7.13:

```java
public class CreationTest {
    static {
        DFA.staticInitStarted(MethodStartSample.class);
        try {
        } finally {
            DFA.methodEnded();
        }
    }

    public CreationTest() {
        DFA.constructorStarted(this, CreationTest.class);
        try {
        } finally {
            DFA.methodEnded();
        }
    }
}
```

Listing 7.13: Instrumented code for class and object creation

When a class contains fields that are initialized as shown in Listing 7.14, the meta-model must be informed of this initialization. For static fields this must be done in a static initializer, while for instance fields this is done in a constructor. In the case where there are multiple constructors, the define action must be placed to ensure that it is only called once per object. The code in Listing 7.14 illustrates the define action for the initialized fields.

```java
public class InitializedVariables {
    {
```
private static int statInt = 10;
private int objectInt = 20;

static
{
    DFA.staticInitStarted(InitializedVariables.class);
    try
    { 
        DFA.define("statInt");
    }
    finally
    { 
        DFA.methodEnded();
    }
}

public InitializedVariables()
{
    DFA.constructorStarted(this, InitializedVariables.class);
    try
    { 
        DFA.define("objectInt");
    }
    finally
    { 
        DFA.methodEnded();
    }
}

Listing 7.14: Instrumented code for initialized fields

7.2.3 Targeted Analysis

The source code instrumentation approach presented is able to overcome some of the issues raised by the approach of Boujarwah et al. [2]. Specifically, this addresses the following points:

1. Variables are located within blocks, which are managed in the presence of exceptions.

2. Information related to an object is loaded within the object’s constructor.

3. Static fields are loaded when the class is loaded by the JVM.

4. Object fields are loaded for instrumented classes when constructed. This includes cases where the object is constructed from within uninstrumented code.

5. Methods provide sufficient information to identify themselves correctly.

However, there are two main issues with source code instrumentation. These are:
1. All actions for public fields cannot be tracked unless all of the source code is instrumented.

2. Exceptions and short circuit evaluation of expressions may result in incorrectly reported actions.

These are discussed in more detail in the following sections.

**Public Fields**

Fields which are publicly defined can be accessed from uninstrumented code. Source code instrumentation does not provide any mechanism to address this issue. In order to track the analysis of these fields, all of the source code for the program must be instrumented.

**Short Circuit Evaluation and Exceptions**

Short circuit evaluation of expressions is a known issue for source code instrumentation. Some of the existing approaches, including [6, 8, 17], place instrumented probes beside each statement. Where the language supports short circuit condition evaluation these conditions must be decomposed to ensure that the actions reported actually occurred during execution. Another approach, proposed by Price [24], replaces variable access with reference and definition functions. As Java does not provide mechanisms to operate with the address of variables, this approach cannot be applied.

Exceptions introduce additional complexities for an approach that uses source code instrumentation. In Java, `RuntimeException` may be thrown in a number of cases, for example `NullPointerException`, and `ArithmeticException`. The code in Listing 6.1 showed a case where an `ArithmeticException` will be thrown. If source code probes are inserted before the statement, as shown in Listing 7.15, the reference of ‘d’ will be incorrectly reported. In the presence of exceptions, almost all statement must be decomposed into individual actions.

**7.3 Byte Code Instrumentation**

Rather than performing source code instrumentation, another approach is to instrument the Java class files. The Java compiler will usually\(^3\) convert the Java source code into the class file.

---

\(^3\)The Java specifications indicate that a Java compiler need not compile Java source code to Java class files [14, 21].
format as specified in [21]. Rather than instrumenting the source code the resulting Java byte code can be modified. The Java byte code is effectively a stack based programming language. If we introduce statements into the byte code that have no effect on the stack, we can modify the byte code without changing the semantics of the program.

Actions which perform loading of values onto the operand stack, such as `iload` that loads a variable onto the stack, are preceded by operations which indicate the reference of the variable. Actions which store values into variables, such as `istore` that stores a value in a local variable, are preceded by operations which indicate the definition action. In both of these cases, the inserted code must not affect the operand stack. The generation of the modified byte code can be performed either by a customized compiler, or by using an existing compiler and modifying its resulting byte code.

### 7.3.1 Identifying Objects and Probe Methods

Byte code and source code instrumentation both use information available in the Java runtime. As a result, the identification of objects and methods used on the DFA class are the same for byte code instrumentation as they where for source code instrumentation. Essentially, byte code instrumentation allows for the reporting of the actions performed to be closer to the actual action, thereby avoiding the issue related to exceptions and short circuit evaluation found with source code instrumentation.
7.3.2 Example Byte Code Instrumentation

The code shown in Listing 6.1 illustrated a case where source code instrumentation requires decomposition to correctly handle exceptions. The byte code generated for the instrumented version of this is shown in Listing 7.16. As can be seen from this listing, the call to reference for variable ‘d’, on lines 149 and 150, is much earlier than its loading, on line 159. The byte code instrumented version is able to place the reference call closer to the actual reference, as shown in Listing 7.17. In this case the call to reference for variable ‘d’ occurs immediately before the loading of the variable, lines 154 to 159.

```
134 ldc "a"
136 invokevirtual swin.ddfa.DFA::void reference(java.lang.String)
139 ldc "b"
141 invokevirtual swin.ddfa.DFA::void reference(java.lang.String)
144 ldc "c"
146 invokevirtual swin.ddfa.DFA::void reference(java.lang.String)
149 ldc "d"
151 invokevirtual swin.ddfa.DFA::void reference(java.lang.String)
154 iload 1
155 iload 2
156 iload 3
157 isub
158 idiv
159 iload 4
```

Listing 7.16: Java byte code of Listing 7.15

```
134 ldc "a"
136 invokevirtual swin.ddfa.DFA::void reference(java.lang.String)
139 iload 1
140 ldc "b"
142 invokevirtual swin.ddfa.DFA::void reference(java.lang.String)
145 iload 2
146 ldc "c"
148 invokevirtual swin.ddfa.DFA::void reference(java.lang.String)
151 iload 3
152 isub
153 idiv
154 ldc "d"
156 invokevirtual swin.ddfa.DFA::void reference(java.lang.String)
159 iload 4
```

Listing 7.17: Java byte code instrumented version of Listing 6.1

7.3.3 Targeted Analysis

Using byte code instrumentation, the issues related to short circuit evaluation and exceptions are avoided. However, byte code instrumentation approaches can only guarantee to fully track the access and/or modification of public fields by violating targeted analysis. It is possible
to instrument all Java class files with instrumented probes, and so libraries and third party components can be instrumented. This, however, will result in anomalies being reported for variables that the developer has no control over. This will negatively impact on the ability to analyse the results of this approach. Targeted analysis is gained using the compiler approach, but as it relies on the program’s source code, it provides no better coverage than the source code instrumentation approaches, in relation to public fields.

7.4 Java Platform Debugger

The Java Platform Debugger Architecture [32] provides three interfaces related to performing debugging tasks using the Java Virtual Machine. By using the Java Debug Interface (JDI) API [31] it is possible to add watch points for access and modification of instance and class fields.

The JDI provides a number of events that can be registered with the debugger. This includes a number of watch point events, via the \texttt{com.sun.jdi.event.WatchpointEvent} interface. Two variants of this interface exist, being the \texttt{AccessWatchpointEvent}, and \texttt{ModificationWatchpointEvent}. The \texttt{AccessWatchpointEvent} is raised when a field of a class or object is read. The \texttt{ModificationWatchpointEvent} is raised when the value of a field is modified. Using these two events it is possible to track all actions that occur on publicly declared fields.

7.4.1 Identifying Objects

Object identification is directly addressed by the JPDA. The \texttt{WatchpointEvent} has an associated \texttt{com.sun.jdi.ObjectReference}, which represents the object whose field was accessed or modified. The \texttt{ObjectReference} interface has a \texttt{uniqueID} method that returns a guaranteed unique identifier for the object. This identifier can therefore be used to track the object within the meta-model.

7.4.2 Issues with JPDA

The JPDA does not provide the facilities for watches to be added to variables local to methods, or to the access or modification of elements in an array. By performing some source code transformation it is possible for the Java Debug Interface [31] to analyze local variables. Wrapper classes can be introduced to provide an object wrapper for local variables. By converting all local variables into objects, appropriate watch points can be added to these instance variables.
All use of the variable within the method are then converted to use of the instance variable. The code in Figure 7.1 shows the original source on the left, with the resulting instrumented source on the right.

```java
1  int aMethod ()
2  {
3      int x;
4      x = 100;
5      return x;
6  }
```

```java
1  int aMethod ()
2  {
3      DfaIntWrapper x = new DfaIntWrapper ();
4      x.value = 100;
5      return x.value;
6  }
```

```java
7 8 9
10  class DfaIntWrapper
11  {
12      public int value;
13  }
```

Table 7.1: Transforming local variables into fields

The other problem still remains, the Java Debug Interface [31] does not allow watch points to be added to individual array elements. Although source code transformation can be provided to track array element usage within the instrumented code, this will again not extend to uninstrumented code. For example the `java.utils.Arrays` class [30] provides methods which take arrays as parameters, e.g. methods for searching, sorting, and filling arrays. Any use of array elements within these methods cannot be tracked by the debugger.

### 7.4.3 Targeted Analysis

This approach comes the closest to meeting all of the requirements from 4.3 for complete data flow analysis. It is capable of both complete analysis of public fields and targeted analysis of a program in the absence of arrays. However, it is unable to perform complete analysis of arrays that are passed to uninstrumented code. There is also a significant overhead to enable the analysis of variables local to methods, where each local variable must be replaced with an object in order to attach the watch.

### 7.5 Summary and Recommendations

Using source code instrumentation, probes are inserted into the source code of the program under analysis. Exceptions and short circuit evaluation cause particular problems for this approach. Byte code transformation inserts the probes directly into the byte code of the java class.
files. This avoids the issues with exceptions and short circuit evaluation that exist with source code instrumentation. In both of these cases, publicly declared fields cannot be fully tracked without instrumenting the entire program, including associated libraries. Public fields can be addressed by using the Java Platform Debugger Architecture, at the expense of significant overhead to analyse local variables. In all cases the use of array elements in uninstrumented code cannot be tracked.

Of the approaches presented, byte code transformation provides the most practical solution. This approach provides the ability to place the probes closer to the location of the action than source code instrumentation, and does not have the overheads related to the use of the Java Platform Debugger Architecture. The JPDA provides better coverage of public fields, but as it is generally accepted that the use of public fields is bad practice, this is not of significant importance. The inability to track arrays in uninstrumented code is of concern, but this issue cannot be addressed using any of these approaches, and remains as something for future work.
Part IV

Conclusions and Future Work
Chapter 8

Conclusions

In this thesis a new approach for managing and collecting information for data flow analysis has been presented. The previous approaches to dynamic data flow analysis of object oriented programs by Chen and Low [10], and Boujarwah et al. [2] were discussed in turns of their modeling of data flow information, Chapter 3, and their instrumented probes, Chapter 6. In order to evaluate these existing approaches, and to guide the creation of a new approach Chapter 2 defined the following set of requirements:

1. As a minimum the approach must allow the tracking of actions for the definition, reference and destruction of all variables under investigation.

2. The approach must be able to handle any type of variable, independent of scope, type, or visibility.

3. The approach must support targeted analysis of source, thereby allowing analysis of individual parts of a system, and also allowing analysis of systems that use third party components.

4. The output generated by the approach must enable programmers to identify the location and type of any anomalies produced.

5. The approach must enable automated analysis.

The existing approaches where evaluated against this set, and two main requirements for new approaches were identified, specifically:

1. A technique to identify and locate state variables must be provided.
2. Targeted analysis must be addressed.

In order to address these requirements, Chapter 4 presented a new, object oriented, approach to performing dynamic data flow analysis. Using this approach a meta-model of the languages structure is defined to maintain the data flow information. To illustrate this model, Chapter 5 defined a meta-model for a simplified set of the Java language. The meta-model developed required certain information to be collected from executing the program under analysis. Three different approaches for collecting this information for the Java model are detailed in Chapter 7.

This thesis does not address the automation of analysis for the techniques presented. The technique itself allows for automation of program instrumentation, however automation of analysis requires significantly more work to address. This is discussed further in the future work chapter which follows.

8.1 Model for Data Flow Analysis

The meta-model developed in Chapter 4 contains elements to store and manage information for data flow analysis. This model is made up of three main parts, State Identifiers, Scoping Components, and Scoping Component Containers. The State Identifiers are used to store information related to data items within the program, and to manage the data flow analysis for this data item. Scoping Components are used to represent the scopes within which data items may exist within the program. For example, these would include elements such as objects, and methods. Finally Scoping Component Containers are used to manage access to individual Scoping Components, examples of Scoping Component Containers include threads, and heap space in Java.

The meta-meta-model is illustrated in Figure 8.1. The Scoping Component Containers manage a collection of Scoping Components, that in turn contain State Identifiers. When an action occurs within the program under analysis, the model is informed and uses its current representation to locate the associated State Identifier. For example, with direct variable access, the Scoping Component Container representing the method stack would be informed of the action. The Scoping Component Container then delegates the locating of the variable to the appropriate Scoping Component, in this case representing the method on the top of the stack. The Scoping Component may then delegate the search onto other Scoping Component, such
as blocks within the method. Each Scoping Component will try to locate the appropriate State Identifier for the data item. The exact nature of the search is determined by the language’s scoping rules. When the State Identifier is located, it will be informed of the action that has occurred and can update its state as it sees fit. In the case of pointers the State Identifier may need to pass the action to another State Identifier.

![Diagram of Meta Meta Model](image)

Figure 8.1: Meta Meta Model defined in Chapter 4

### 8.2 Data Flow Analysis for Java

The Java implementation of the meta-meta-model was presented in Chapter 5. The meta-model developed is shown in Figure 8.2. The meta-model contains Scoping Components for classes and objects, DFA.Class and DFA.Object, methods and blocks, DFA.Block, DFA.Method, and related classes, and a single State Identifier, DFA.Variable. Within the meta-model, arrays are modeled as a kind of object which delegates its actions to its elements. The meta-model also contains Scoping Component Containers to represent the method stack\(^1\), and heap space.

The model defined a number of events that must be communicated to it from the program under analysis. These events include the start of Scoping Components such as methods and blocks, as well as events like object creation, class loading, and variable use. Chapter 7 presented three approaches to performing this; namely, source code instrumentation, byte code transformation, and via the Java Platform Debugger Architecture.

Source code instrumentation uses probes inserted into the source code of the program under analysis. These probes report the events directly to the meta-model. Unfortunately, source code

\(^1\)Multiple threads are ignored.
Figure 8.2: Java data flow analysis classes from Chapter 5

instrumentation is unable to meet the targeted analysis requirements for public fields, and can incorrectly report actions due to exceptions. The Java Platform Debugger Architecture provides the facilities to fully track the access and modification of public fields, but is unable to perform the analysis for local variables and individual array elements without significant overheads. Byte code transformation provides the most practical solution, but is still unable to meet all of the specified requirements. Byte code transformation is able to avoid most of the issues of source code instrumentation but cannot fully track public fields, and array elements.
8.3 Comprehensive Analysis

An implementation of the approach presented here will be able to provide comprehensive analysis for the following elements of a Java program:

1. Private instance fields

2. Local variables

3. Fields declared as public, protected, or with default scope, where all uses of these fields occur within instrumented code.

4. Arrays, where all uses of the arrays occur within instrumented code.

This will offer sufficiently comprehensive analysis for the majority Java applications, and represents the most comprehensive analysis using the approaches discussed.
Chapter 9

Future Work

This thesis has presented a framework within which data flow information can be collected for the analysis of a program. There are several key areas that require further research related to this work. These include further developing the model presented, defining new models for other languages, and examining ways to effectively use the information gained from this analysis. In addition to these future works, further research needs to be conducted in order to evaluate the effectiveness of this approach as compared to other techniques, such as static data flow analysis.

The main hindrance to practical dynamic data flow analysis is the ability to perform a comprehensive targeted analysis. Attempts to address this in this thesis have been unable to determine a complete solution for Java. Within this thesis the bounds of what can and cannot be performed have been clearly stated. In order for this to be more applicable these boundaries need to be further tested. For example, with greater support built into the JPDA a full comprehensive analysis becomes possible.

A mechanism to collate and evaluate the information gained from performing this analysis needs to be created. Existing approaches to dynamic data flow analysis have found that the quantity of information gained using these techniques has been extensive. As a result some form of aggregate view needs to be developed. With object oriented programs, classes and methods, as distinct from objects and methods, provide an interesting possibility as a point of aggregation. For example, for all of the objects of a certain class, what percentage had anomalies related to a given field? At a method level, information related to the number of times a path is executed versus the number of times an anomaly occurs on the path can also provide an insight into where more attention should be paid. Research in this area will be
beneficial to this and other approaches.

The Java model presented in this thesis contains a number of simplifications. These simplifications need to be addressed and a complete meta-model for Java, incorporating inner classes, anonymous classes and concurrency needs to be presented. This will require a significant contribution in relation to performing data flow analysis in concurrent environments.

The general applicability of the meta-meta-model needs to be evaluated by defining a meta-model for a language other than Java. The .NET framework [13] provides an interesting area for this evaluation, as a .NET meta-model would need to be applicable to multiple languages, supporting at least C# and Visual Basic.NET. The application of the meta-meta-model to a non object oriented language would also be of interest as objects provide a natural way of tracking software elements for this kind of analysis.
Part V

Appendices
Appendix A

Source Code Examples
A.1 C++ Source Code Sample

```cpp
#include <iostream>

class Point {
  protected:
    int x, y;

  public:
    Point(int x, int y) {
      this->x = x;
      this->y = y;
    }

    friend ostream& operator<<(ostream& os, Point& p);
};

class MovablePoint : public Point {
  private:
    int delta;

  public:
    MovablePoint(int x, int y, int delta) : Point(x,y) {
      this->delta = delta;
    }

    void move() {
      x += rand() % delta - delta / 2;
      y += rand() % delta - delta / 2;
    }

    ostream& operator<<(ostream& os, Point& p) {
      return os << "X: " << p.x << " Y: " << p.y << endl;
    }

  int main() {
    // Initialize the random number generator with the current time.
    time_t tnow;
    time(&tnow); // Get the time
    srand(tnow); // Set the random seed

    // Create, move, and print the point...
    MovablePoint p(10,100,10);
    p.move();
    cout << p;
  }
```

Listing A.1: C++ Point source code.
A.2  Java Source Code Sample

class Particle
{
  private int x = 0;
  private int y;
  private static final java.util.Random rng = new java.util.Random();

  public Particle(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  public int getX()
  {
    return x;
  }

  public int getY()
  {
    return y;
  }

  public void move()
  {
    x += rng.nextInt(10) - 5;
    y += rng.nextInt(20) - 10;
  }

  public String toString()
  {
    String s = "X: " + x + " Y: " + y;
    return s;
  }
}

class ParticleTest
{
  public static void main(String[] args)
  {
    Particle p = new Particle(100, 100);
    p.move();
    System.out.println(p.toString());
    Particle p1 = p;
    int p1x = p1.getX();
    System.out.println("p1.x = " + p1x);
  }
}

Listing A.2: Java Point source code.
Appendix B

Java Model Classes
B.1 DFA_Variable

![DFA_Variable Class Diagram](image)

Figure B.1: DFA_Variable Class

<table>
<thead>
<tr>
<th>Variable</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>This integer contains the identifier of the variable's current state.</td>
</tr>
<tr>
<td>name</td>
<td>The name instance variable contains the name of the variable.</td>
</tr>
<tr>
<td>instanceVariable</td>
<td>This indicates that this object refers to an instance variable. This is used in the reporting of the anomaly.</td>
</tr>
<tr>
<td>instanceOf</td>
<td>In the case of instance variables, this reference variable refers to the DFA_Object that the variable exists within.</td>
</tr>
<tr>
<td>declaredIn</td>
<td>In the case of instance variables, this variable indicates the class in which the variable was declared.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>processAction</td>
<td>This method indicates that the DFA_Object is to process an action.</td>
</tr>
<tr>
<td>reportAnomaly</td>
<td>The reportAnomaly method is called by processAction when the anomaly state is entered. This reports the anomaly to the user.</td>
</tr>
</tbody>
</table>

Table B.1: Methods of DFA_Variable
B.2 DFA_Object

Table B.2: Methods of DFA_Object

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>getField</td>
<td>This method searches for a variable that matches the passed in string.</td>
</tr>
<tr>
<td>DFA_Object()</td>
<td>This constructor is used by child classes to avoid the loading of instance variables.</td>
</tr>
<tr>
<td>DFA_Object(Object)</td>
<td>This constructor loads the DFA_Object with DFA_Variables for the specified object.</td>
</tr>
<tr>
<td>processAction</td>
<td>This method indicates that the DFA_Object is to process an action.</td>
</tr>
<tr>
<td>doProcessAction</td>
<td>This method processes the action for DFA_Obejcts.</td>
</tr>
<tr>
<td>destroyed</td>
<td>Indicates that the object has been destroyed. This undefines all variables for this object.</td>
</tr>
<tr>
<td>addInstanceVariable</td>
<td>Adds a variable to the object. This is called by the constructor, and by child classes.</td>
</tr>
<tr>
<td>getName</td>
<td>Returns the name of the class used to instantiate this object.</td>
</tr>
<tr>
<td>toString</td>
<td>Returns a string representation of the object used for reporting.</td>
</tr>
</tbody>
</table>

These methods are described in more detail in Section 5.4 Searching in the Java Model.
B.3 DFA_Class

![DFA Class Diagram](image)

Figure B.3: DFA.Class

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFA.Class</td>
<td>This constructor loads the DFA.Class with DFA.Variables for the specified class. This uses DFA.Object’s default constructor to avoid loading the instance fields.</td>
</tr>
</tbody>
</table>

Table B.3: Methods of DFA.Class
B.4 DFA_Array

The constructor creates the DFA_Array and creates DFA_Variable objects for the array’s elements. Arrays with more than one dimension are represented as arrays of arrays, as such a multi-dimensional array will have multiple DFA_Array objects.

processArrayAction
This method calls processAction on the DFA_Variable object at the specified index.

processAction
An array is unable to process actions in the same way as objects, as a result this method throws a RuntimeException.

destroyed
This indicates that the array has been destroyed, resulting in an undefined action for each of its elements.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFA_Array</td>
<td>The constructor creates the DFA_Array and creates DFA_Variable objects for the array’s elements. Arrays with more than one dimension are represented as arrays of arrays, as such a multi-dimensional array will have multiple DFA_Array objects.</td>
</tr>
<tr>
<td>processArrayAction</td>
<td>This method calls processAction on the DFA_Variable object at the specified index.</td>
</tr>
<tr>
<td>processAction</td>
<td>An array is unable to process actions in the same way as objects, as a result this method throws a RuntimeException.</td>
</tr>
<tr>
<td>destroyed</td>
<td>This indicates that the array has been destroyed, resulting in an undefined action for each of its elements.</td>
</tr>
</tbody>
</table>
B.5 DFA_Method

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFA_Method</td>
<td>The three constructor versions for the DFA_Method are used to create different types of methods. The default constructor is protected and is only used by child classes. The version that accepts the method’s name is used to create standard methods. The other version is used to create constructors, and therefore does not require the name of the method.</td>
</tr>
<tr>
<td>addVariable</td>
<td>This indicates that a new local variable has been declared and must be added to the method’s current block.</td>
</tr>
<tr>
<td>processAction²</td>
<td>This method is called to indicate that an action has occurred and must be passed to the appropriate DFA_Variable.</td>
</tr>
<tr>
<td>processSuperAction²</td>
<td>This method is used in the place of processAction where a field is accessed via the super keyword.</td>
</tr>
<tr>
<td>endMethod</td>
<td>Indicates that the method has terminated. All DFA_Variable objects declared in this method will receive undefined events.</td>
</tr>
<tr>
<td>addBlock</td>
<td>Indicates that a new block has been entered in the program. This creates a new DFA_Block and sets it as the current block.</td>
</tr>
<tr>
<td>endBlock</td>
<td>This method is called when the block ends. This changes the method’s current block and undefines all DFA_Variable objects contained with the block.</td>
</tr>
<tr>
<td>catchFinallyBlock</td>
<td>This method is used when a catch or finally block is entered in the program. This uses the current method and block information to synchronize the model and code.</td>
</tr>
<tr>
<td>methodString</td>
<td>This method returns the name of the method. For constructors this is set to</td>
</tr>
</tbody>
</table>

²These methods are described in more detail in Section 5.4 Searching in the Java Model.
B.6 DFA_Block

This constructor creates the DFA_Block and initializes the previous block and block identifier fields.

This method adds a new local variable object to the block. The variable is identified by name.

This method attempts to process the action performed upon a variable. This returns true if a variable was found and the action processed.

This ends the block, undefining any local variables declared within it.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFA_Block</td>
<td>This constructor creates the DFA_Block and initializes the previous block and block identifier fields.</td>
</tr>
<tr>
<td>addVariable</td>
<td>This method adds a new local variable object to the block. The variable is identified by name.</td>
</tr>
<tr>
<td>processAction</td>
<td>This method attempts to process the action performed upon a variable. This returns true if a variable was found and the action processed.</td>
</tr>
<tr>
<td>endBlock</td>
<td>This ends the block, undefining any local variables declared within it.</td>
</tr>
</tbody>
</table>

Table B.6: Methods of DFA_Block
B.7 DFA_StaticInitializer

![DFA_StaticInitializer Diagram]

Figure B.7: Static Initializer Scoping Component

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFA_StaticInitializer</td>
<td>This constructor creates the DFA_StaticInitializer, no method or parameter names can be passed to this constructor.</td>
</tr>
<tr>
<td>methodString</td>
<td>This method returns the name of the static initializer as ‘&lt;clinit&gt;’.</td>
</tr>
</tbody>
</table>

Table B.7: Methods of DFA_StaticInitializer
B.8 HeapSpace

![Heap Space class diagram]

Figure B.8: Heap Space class

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeapSpace</td>
<td>The constructor initializes the collection</td>
</tr>
<tr>
<td>add</td>
<td>The add method adds a new DFA_Object to the collection.</td>
</tr>
<tr>
<td>remove</td>
<td>This method is used to remove the object from the HeapSpace once it is garbage collected.</td>
</tr>
<tr>
<td>getDFAObjectFor</td>
<td>The getDFAObjectFor method returns DFA_Object instances associated with the specified key object.</td>
</tr>
<tr>
<td>shutdown</td>
<td>Indicates that the runtime is shutting down. This ensures that all objects receive their destroyed message upon program termination.</td>
</tr>
</tbody>
</table>

Table B.8: Methods of HeapSpace
B.9 MethodStack

![Method Stack class image]

Figure B.9: Method Stack class

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>addMethod(DFA_Method)</td>
<td>This method adds the DFA_Method object and to the top of the stack.</td>
</tr>
<tr>
<td>addMethod(...)</td>
<td>This method constructs a new DFA_Method object and adds it to the top</td>
</tr>
<tr>
<td></td>
<td>of the stack. This method is used for both static and instance methods.</td>
</tr>
<tr>
<td>addConstructor</td>
<td>This method constructs a new DFA_Method object and adds it to the top</td>
</tr>
<tr>
<td></td>
<td>of the stack. This method is used only for constructors.</td>
</tr>
<tr>
<td>addStaticInitializer</td>
<td>This method constructs a new DFA_StaticInitializer object and adds it</td>
</tr>
<tr>
<td></td>
<td>to the top of the stack.</td>
</tr>
<tr>
<td>methodReturned</td>
<td>This method is used to remove indicate that the method on top of the</td>
</tr>
<tr>
<td></td>
<td>MethodStack has terminated. This ends the method and removes it from</td>
</tr>
<tr>
<td></td>
<td>the stack.</td>
</tr>
<tr>
<td>startBlock</td>
<td>This indicates to the current method that a new block has started.</td>
</tr>
<tr>
<td>catchBlock</td>
<td>The catchBlock method is called when a catch or finally block is entered</td>
</tr>
<tr>
<td></td>
<td>in the program. This is passed to the current method that ensures that</td>
</tr>
<tr>
<td></td>
<td>the associated block is set as the current block.</td>
</tr>
<tr>
<td>endBlock</td>
<td>This indicates to the current method that its current block has termi-</td>
</tr>
<tr>
<td></td>
<td>nated.</td>
</tr>
<tr>
<td>processAction$^3$</td>
<td>This method is called to indicate that an action has occurred and must</td>
</tr>
<tr>
<td></td>
<td>be passed to the appropriate DFA_Variable.</td>
</tr>
<tr>
<td>processSuperAction$^3$</td>
<td>This method is used in the place of processAction where a field is</td>
</tr>
<tr>
<td></td>
<td>accessed via the super keyword.</td>
</tr>
<tr>
<td>processArrayAction$^3$</td>
<td>This method is used in the place of processAction for accessing of</td>
</tr>
<tr>
<td></td>
<td>array elements.</td>
</tr>
</tbody>
</table>

Table B.9: Methods of MethodStack

---

$^3$These methods are described in more detail in Section 5.4 Searching in the Java Model.
Bibliography


