

# An Object Oriented Approach towards Dynamic Data Flow Analysis

A. Cain, T.Y. Chen, D.D. Grant, F.-C. Kuo\* and J.-G. Schneider

Faculty of Information and Communication Technologies  
Swinburne University of Technology  
P.O. Box 218, Hawthorn, Victoria 3122, Australia  
{acain,tchen,dgrant,dkuo,jschneider}@ict.swin.edu.au

## Abstract

*Dynamic data flow analysis is a testing technique that has been successfully used for many procedural programming languages. However, for Object-Oriented (OO) programs, previous investigations have still followed a data-oriented approach to keep track of the state information for various data elements. This paper proposes an OO approach to perform dynamic data flow analysis for OO programs. In this approach, a meta-model of an OO program's runtime structure is constructed to manage the data flow analysis for the program. An implementation of the model for the Java language is presented, illustrating the practicality and effectiveness of this innovative approach.*

**Keywords:** *Dynamic Data Flow Analysis, Meta-level Programming, Object-Oriented Programs, Program Instrumentation*

## 1. Introduction

*Data flow analysis* is a testing method for detecting improper use of data in programs [8]. This can either be done 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. Dynamic and static analysis are complementary strategies [6].

In conventional dynamic data flow analysis [8], there are three basic *actions* that can be performed on a variable, namely *define*, *reference* and *undefine*. 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 it has not yet been assigned any value, its value is destroyed or it goes out of scope. Data flow

anomalies represent improper sequences of actions performed on a data element. Three data flow anomalies exist, namely, *define-define*, *undefine-reference* and *define-undefine*. 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, then this indicates a *undefine-reference* anomaly. The define-undefine anomaly indicates that the data element's value has been defined but not used before the value is destroyed. Huang [8] proposed the tracking of *states* via a state machine instead of the tracking of actions. In its basic form, there are four states, namely *Defined*, *Undefined*, *Referenced* and *Anomaly*. The three actions act as triggers to state transitions and their effects are illustrated as a state machine in Figure 1. Entering into the Anomaly state indicates the occurrence of an anomaly.

Although some methodologies have been developed to handle dynamic data flow analysis for Object-Oriented (OO) programs, some problems still remain. All the existing methodologies have taken a data-oriented approach to performing data flow analysis. Object is the key element for OO programming languages. Each object consists of data and functions, and these data and functions could be inherited from other objects. The main issue with dynamic data flow analysis for OO programs is the ability to distinguish the actions on an object's variables from all other objects in the program. The existing data-oriented approach does not provide a natural mapping to OO languages. Neither does it take advantage of OO constructs. An OO approach to dynamic data flow analysis will provide a more straightforward and natural solution for OO programs. In this paper, we will present this new approach for detecting data anomalies within OO programs by means of meta-level abstractions [9].

The rest of the paper is organized as follows: Sec-

---

\* Corresponding author

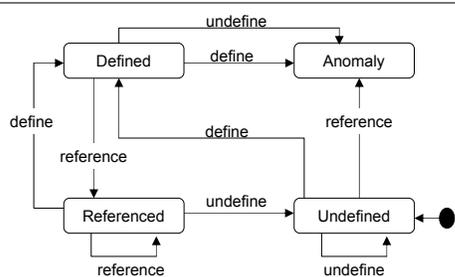


Figure 1. A state machine for data flow analysis

tion 2 presents a discussion of previous studies on performing data flow analysis. In Section 3, we define an OO approach to performing dynamic data flow analysis. Section 4 illustrates the OO approach using Java, followed by a case study in Section 5. We conclude this paper and outline future directions in Section 6.

## 2. Related Work

Dynamic data flow analysis has been performed for various procedural [3, 4, 8] and OO [1, 5] programming languages. All of these existing approaches follow the basic approach taken by Huang [8], who proposed to detect data flow anomalies via program instrumentation. In Huang’s method [8], each variable has an explicitly declared *state variable* whose name is defined by adding a reserved “prefix” to the corresponding variable’s name. State variables named in such a way are called *explicit* state variables. Variables and their state variables are then *linked* through their names. Price and Poole [12], and Chen and Low [5] proposed to use the notion of *implicit* state variables. In their approach, the “memory location” and “size” of the variable are used as the identification keys to track the actions related to the variables. Whilst this technique is sufficient for C++, it requires a mechanism to access the memory location of a variable, which may not be available in other OO programming languages (e.g., Java).

Boujarwah *et al.* [1] developed an approach for analysing Java programs. This approach uses explicit data names to track the data elements within a program, and identifies instance variables, local variables and class variables via a “combination of *identifiers*”. For example, an instance variable is identified by a combination of the instance variable’s name, object name, and the class in which the variable was declared. This approach requires a significant amount of work to maintain the object identifiers during execution. Cain *et al.* proposed several techniques for extracting data flow information from Java programs [2] including source code instrumentation, byte code instru-

mentation, and instrumentation using the Java Platform Debugger Architecture [11].

## 3. An OO Approach

Different from other methodologies, we propose an OO approach to dynamic data flow analysis, which provides a more straightforward and natural solution for OO programs. The main component of our approach is a meta-model of a program’s runtime structure. This model is responsible for managing the data flow analysis for the OO program under inspection.

The meta-model contains elements that represent the *scoping components* of the language, such as classes, objects, methods and blocks. These elements are the *variable containers* or containers of other scoping components. The meta-model relies on appropriate notifications (such as creation and destruction of scoping components) from the program under inspection, in order to create a correct “meta-level representation” of the program’s runtime structure. The meta-model is to observe the actions that are performed upon the variables within the instrumented program. Each variable in the instrumented program has an associated object in the meta-model, which keeps track of all actions performed upon it. The *scoping rules* of the language are implemented inside the meta-model, so when an action is performed on a variable in the instrumented program, the meta-model can locate the corresponding *meta-object* for this variable, and update its state accordingly.

To implement the OO approach for dynamic data flow analysis, the following steps must be undertaken.

### Step 1. Variable analysis

Different state machines for data flow analysis are proposed [1, 3, 4, 8]. The meta-model implements one of these state machines to perform analysis. A *Variable* class is designed to implement this functionality. When an action is performed upon a variable in the instrumented program, its corresponding *Variable* object in the meta-model is informed of the action and performs any required transitions on the variable’s state, as defined by the state machine.

### Step 2. Modelling scoping structures

The meta-model contains classes designed to represent the scoping components of the language. These elements within the meta-model must reflect their language counterparts as both containers for variables, and containers for other scoping components. The meta-model is an abstraction, or observation, of the runtime structure of the instrumented program.

### Step 3. Locating variables inside the model

When the instrumented program notifies the meta-model about an action performed on a variable, the meta-model makes use of the scoping rules to find the corresponding `Variable` object within itself, and updates the state of that object.

**Step 4. Interacting with the meta-model**

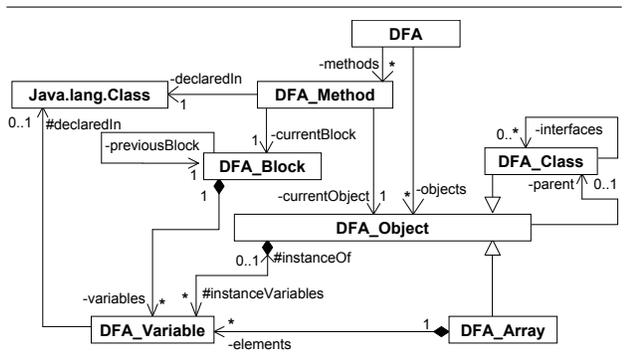
The program under analysis reports certain events to the meta-model to allow for creating a runtime structure representation and tracking the data usage of the program. These events include at least the following:

1. Creation and destruction of scoping components, including (a) creation and destruction of objects, (b) start and end of methods, and (c) start and end of blocks
2. Creation and destruction of variables, and actions performed upon them

For example, when a variable `x` declaration in the instrumented program is reported, a new `Variable` object is constructed and added to the meta-model. The actions performed upon `x` are used to trigger the state changes for the `Variable` object of `x`. When the meta-model is informed of the destruction of `x`, the corresponding `Variable` object is removed from the model.

**4. Illustration Using Java**

In this section, we discuss the OO approach of dynamic data analysis in the context of Java programs. This language was chosen because it is OO, supports only pass by value, and provides abstractions for runtime introspection. To simplify this illustration, we will ignore inner and anonymous classes as well as threads and concurrency. This allows us to concentrate on the process of applying the OO technique rather than on dealing with the language specific details. We have implemented a meta-model for this study. Figure 2 outlines the static structure of the meta-model.



**Figure 2. Static structure of the meta-model**

**4.1. Variable Analysis for Java**

For the purpose of illustration, we use the basic state machine defined by [8], as shown in Figure 1. The corresponding set of states and transitions is implemented in the `DFA_Variable` class. This class could be extended to incorporate more advanced state management, for example, the alternative state diagram presented by Huang in [8], or the Java specific version discussed in [1]. When an anomaly occurs, variable identification and scoping information are used to report the location and type of anomaly.

**4.2. Scoping Structures for Java**

There are three categories of variables in Java [7], namely *local variables*, *instance variables*, and *class variables* (also called *static variables*). Local variables, including method parameters, have a block scope and can be accessed only from the current block and any child blocks. Instance variables can be accessed from within the object or via an object reference. Access to instance variables is controlled by a scope modifier: being either *public*, *protected* or *private*, or default when no explicit modifier is given. Static variables can be accessed from any object created from the class or any child class, and also via a reference to the class itself. Access to static variables is also controlled by scope modifiers in the same way as for instance variables. *Constants* defined within an interface can be accessed in the same fashion as static variables, and are also accessible from any class which implements the interface. Java includes arrays which are reference types containing an indexed set of values. The values within the array are also variables. To represent these scoping components of Java, the meta-model includes `DFA_Block`, `DFA_Method`, `DFA_Object`, `DFA_Class` and `DFA_Array`.

**4.3. Locating DFA\_Variables**

With the static scoping structure of a Java program defined, the remainder of the meta-model’s functionality is concerned with using this structure to locate variables. Information about the actions performed upon variables of the program under inspection needs to be routed through the model to the corresponding `DFA_Variable` object. It is implemented according to the scoping rules shown in the Java language specification [7] with the support of the runtime introspection services and reflections API of Java. The meta-model implements the scoping rules for Java as follows:

1. Variables are either: (a) local variables to methods, contained within the `DFA_Blocks` of a

DFA\_Method, (b) instance variables, accessed via a DFA\_Object element, (c) static variables, accessed via the DFA\_Object, that is, the super class of a DFA\_Class object, (d) interface constants, accessed via a DFA\_Object's interface, or (e) array elements, contained within a DFA\_Array

2. A method has a current block, which in turn has previous blocks
3. A method is executed on an object or class
4. An object is created from a class, implemented via the parent relationship
5. A class has a single parent class, or no parent in the case of `java.lang.Object`, also implemented via the parent relationship
6. An class may implement a number of interfaces

#### 4.4. Interacting with the Meta-Model

The program under inspection needs to interact with the meta-model. Technically speaking, the model provides a global point of access via the class of DFA (acronym for "Data Flow analyser"). The DFA class is responsible for managing access to the meta-model elements. To efficiently managing the access, the DFA class keeps a *stack* of DFA\_Methods representing a list of the methods being executed in the instrumented program. Furthermore, the DFA class maintains a *hash table* of DFA\_Objects and DFA\_Classes, so if an instance variable, static variable or constant is directly referenced in the program, the DFA class can directly inform the related meta-object of this *reference* action.

The DFA class provides the following methods to handle events related to the program under inspection.

1. Actions performed upon variables, including the define and reference methods, as well as `superDefine` and `superReference` methods for handling variables of the parent classes
2. Creation of local variables, `declare`
3. Start of a method, including: (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, `catch-FinallyStarted`.

The DFA class is also responsible for cleaning up the memory for the meta-objects being created during the analysis process, once it receives messages from the instrumented program about the destruction of the variables, blocks, methods, objects, classes and arrays. The following list some of DFA's methods for this service.

1. Disposing of DFA\_Object objects: `destroyObject` method called from the objects `finalize` method
2. Disposing of the meta-objects upon program termination: `shutdown` method which is called via a shutdown hook attached to the Java Runtime.

#### 5. Case Study

We applied the developed meta-model to a Java program. The chosen program is a simplified version of the Particle program from [10]. This program creates two particles, moves the particles twice, and prints the new locations of particles after each move. The source code is given in the Appendix. The following outlines the initial execution segment of the data flow analysis.

When this program starts, the Java Virtual Machine (JVM) initially loads the `ParticleTest` class. A static initializer in the `ParticleTest` class informs the meta-model that the class has been loaded. This is done using the `staticInitializerStarted` method of the DFA class. At this time, the model creates a `DFA_Class` object to represent this class. In order to create this `DFA_Class` object, additional `DFA_Class` objects must be created for any parent classes of `ParticleTest`. In this case, a `DFA_Class` object must be created for the `java.lang.Object` class.

Once the `ParticleTest` class is loaded, its static `main` method will be executed (refer to Line 03 of the source code). When this method starts, it executes the `classMethodStarted` method on the DFA class. This method constructs a `DFA_Method` object that references the `ParticleTest`'s `DFA_Class` as the current object. This `DFA_Method` object is added to the *method stack* (that is, the stack maintained inside the DFA class. Refer to Section 4.4). Since the `main` method has a parameter (`args`), a `DFA_Variable` object is created as a local variable of the current block of the `main`'s `DFA_Method` object. In this case, the variable `args` refers to an array, thus a `DFA_Array` object is also created.

In the `main` method, several other variables are declared and objects are created. On Line 05, a variable `pc` is declared, a `ParticleContainer` instance is created, and its reference is stored in the `pc` variable. When the declaration of the `pc` variable occurs, the `declare` method of the DFA class is executed and a `DFA_Variable` object is created as a local variable of the current block in the `main`'s `DFA_Method`. Before creating the `ParticleContainer` instance, the JVM loads the `ParticleContainer`

class, and the meta-model constructs a corresponding `DFA_Class` object. When this `DFA_Class` object is created, a `DFA_Variable` object is created for the static variable, called `MAX_PARTICLES` declared on Line 26. This `DFA_Variable` object is stored in the `DFA_Class`'s `instanceVariables` collection, and its state after executing Line 26 is *Defined*. The creation of the new `ParticleContainer` instance on Line 05 results in a `DFA_Object` being constructed with additional creation of `DFA_Variable` objects for the `particles` and `noParticles` instance variables; and furthermore, the state of `pc` being updated to *Defined* by the corresponding `DFA_Variable` object. Since the `particles` variable refers to an array of `Particle`, a `DFA_Array` object is created, and the state for `particles` is updated to *Defined*. The `MAX_PARTICLES` is referred during the array creation, so its `DFA_Variable`'s state is updated to *Referenced*. The `noParticles` variable is declared without given any value, so its state remains *Undefined*.

When Line 07 is executed, the JVM loads the `Particle` class, and the meta-model constructs a corresponding `DFA_Class` object. This `DFA_Class` object is referenced by the `DFA_Object` of the new created `Particle` instance. This `DFA_Object` will associate with three `DFA_Variable` objects, one for each instance variable: `x`, `y`, and `rng` (see Lines 56-58). No values were assigned to `x` and `y`, so their `DFA_Variable` objects have the state of *Undefined* at that point. Since the `rng` variable refers to a new `java.util.Random` instance, a `DFA_Object` for this instance is created and added to the meta-model, and the state for `rng` is *Defined*.

At the start of the `Particle` constructor execution (Line 60), a new `DFA_Method` object is created and added to the method stack. This `DFA_Method` object references the `DFA_Object` of `Particle` as the current object. Since the constructor has two parameters, `x` and `y`, two new `DFA_Variable` objects are constructed and added to the `DFA_Method`'s local variables. When Line 62 executes, the model receives a message that the '`x`' local variable was referenced; this is done via the probe `DFA.define('x')`. The search for this variable begins with the constructor's `DFA_Method` (as it is the top element of the method stack), and results in the message being passed to the `DFA_Variable` of '`x`' parameter in the constructor. The model then receives a message that the '`this.x`' variable was defined for the current object; this is done via the probe `DFA.define('x', this, Particle)`. This causes the model directly accessing the `DFA_Object` and the message being passed to the instance variable's `DFA_Variable` object. The execution of Lines 63 and 62 will lead to the same analysis, thus we skip the discussion for Line 63. At the end of the execution of the `Particle`'s constructor, a `methodEnded` mes-

sage is sent to the `DFA` class, and the `DFA_Method` object on the top of the method stack is destroyed, resulting in *undefine* actions on the two local variables (parameter `x` and `y` of the `Particle`'s constructor).

Due to the page limit, we are unable to show the entire process of analysis. The meta-model will clean up the memory for the meta-objects being created during the analysis process, once it receives messages from the instrumented program about the destruction of the variables, blocks, methods, objects, classes and arrays.

This study shows that our developed meta-model can effectively detect data anomalies in java programs. For example, if the code on Line 63 of the sample program is changed to `this.x = y;`, this will result in a *define-define* anomaly being reported, and a *undefine-reference* anomaly when the `Particle`'s `move` method is executed (refer to Line 69). Chen and Low [5] summarized data flow anomalies for C++. The anomalies described either directly relate to the C++ language, or generally relate to OO programs. Our model can handle OO-related anomalies reported in [5].

## 6. Conclusions

The main issue with dynamic data flow analysis for OO programs is the ability to distinguish the actions on an object's variables from all other objects in the program. All the existing methodologies have taken a data-oriented approach to performing data flow analysis. The existing data-oriented approach does not provide a natural mapping to OO languages. Neither does it take advantage of OO constructs.

Different from these methodologies, we propose an OO approach to dynamic data flow analysis - a more straightforward and natural solution for OO programs. In this approach, a meta-model is constructed to perform the data flow analysis for the program under inspection. This meta-model includes a `Variable` class that acts as the state machine for a single variable. The meta-model captures the languages' scoping structures as `Variable`'s containers, and implements the scoping rules for locating a `Variable` object in the meta-model from a given point. With the built structure and implemented rules, messages from the instrumented program to the meta-model can be used to perform dynamic data flow analysis. The java version of the meta-model was built in this study to illustrate the implementation of the OO approach to dynamic data flow analysis. This model is capable of performing dynamic data flow analysis for single threaded Java programs. Our study shows that the developed meta-model can detect the OO-related anomalies reported in [5].

Our future work is to develop a complete meta-model for Java, incorporating inner classes, anonymous classes and concurrency. The approach can then be extended to other OO languages, with the possibility of creating a generic meta-model. The .NET framework appears to be one area in which a generic meta-model could be developed to handle multiple languages. Each OO language may require a different model to cater for the specific details of that language and any language dependent features, such as pointers in C/C++ [13], or dynamic scoping in CLOS [14]. We will also look into the distinct features of each OO language for more specific data flow meta-model development.

## References

- [1] A. S. Boujarwah, K. Saleh, and J. Al-Dallal. Dynamic data flow analysis for Java programs. *Information and Software Technology*, 42(11):765–775, Aug. 2000.
- [2] A. Cain, J.-G. Schneider, D. Grant, and T. Y. Chen. Runtime data analysis for java programs. In *Proceedings of ECOOP 2003 Workshop on Advancing the State-of-the-Art in Runtime Inspection (ASARTI 2003)*, 2003.
- [3] F. T. Chan and T. Y. Chen. AIDA – A Dynamic Data Flow Anomaly Detection System for Pascal Programs. *Software Practice and Experience*, 17(3):227–239, 1987.
- [4] T. Y. Chen, H. Kao, M. S. Luk, and W. C. Ying. COD – A Dynamic Data Flow Analysis System for COBOL. *Information and Management*, 12(2):65–72, 1987.
- [5] T. Y. Chen and C. K. Low. Error Detection in C++ through Dynamic Data Flow Analysis. *Software – Concepts and Tools*, 18:1–13, 1997.
- [6] T. Y. Chen and P. C. Poole. Dynamic dataflow analysis. *Information and Software Technology*, 30(8):497–505, Oct. 1988.
- [7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.
- [8] J. C. Huang. Detection of Data Flow Anomaly Through Program Instrumentation. *IEEE Transactions on Software Engineering*, SE-5(3):226–236, May 1979.
- [9] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [10] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 2nd edition, 2000.
- [11] S. Microsystems. Java Platform Debugger Architecture. Available at <http://java.sun.com/j2se/1.4.1/docs/guide/jpda/>.
- [12] D. A. Price. Program Instrumentation for the Detection of Software Anomalies. Master’s thesis, University of Melbourne, Australia, 1985.
- [13] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
- [14] P. H. Winston and B. K. P. Horn. *LISP*. Addison-Wesley, 3rd edition, 1989.

## Appendix: Code of the Particle program

```

01 public class ParticleTest
02 {
03     public static void main(String[] args)
04     {
05         ParticleContainer pc = new ParticleContainer();
06
07         pc.addParticle(new Particle(25, 12));
08         pc.addParticle(new Particle(32, 23));
09
10         for(int j = 0; j < 2; j++)
11         {
12             System.out.println(" Move no " + (j + 1));
13             pc.moveAll();
14
15             for(int i = 0; i < pc.getParticleCount(); i++)
16             {
17                 Particle p = pc.getParticle(i);
18                 System.out.println(" " + (i + 1) + ": " + p);
19             }
20         }
21     }
22 }
23
24 class ParticleContainer
25 {
26     private static final int MAX_PARTICLES = 2;
27     private Particle particles[] = new Particle[MAX_PARTICLES];
28     private int noParticles;
29
30     public int getParticleCount()
31     {
32         return noParticles;
33     }
34
35     public void moveAll()
36     {
37         for(int i = 0; i < noParticles; i++)
38         {
39             getParticle(i).move();
40         }
41     }
42
43     public Particle getParticle(int index) throws IndexOutOfBoundsException
44     {
45         return particles[index];
46     }
47
48     public void addParticle(Particle p) throws IndexOutOfBoundsException
49     {
50         particles[noParticles++] = p;
51     }
52 }
53
54 class Particle
55 {
56     private int x;
57     private int y;
58     private final java.util.Random rng = new java.util.Random();
59
60     public Particle(int x, int y)
61     {
62         this.x = x;
63         this.y = y;
64     }
65
66     public void move()
67     {
68         x += rng.nextInt(10) - 5;
69         y += rng.nextInt(20) - 10;
70     }
71
72     public String toString()
73     {
74         return "X:" + x + " Y:" + y;
75     }
76 }

```