Volume 6, Issue 3, March 2016 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Approaches towards Dynamic Data Flow Analysis in Dot Net Programming

Pankaj Patel
M.Tech Scholar,
India

Prof. Pritesh Jain
RGPV, PCST Indore,
India

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

Keywords –DFA, Testing, OO, Inheritance, Modeling

I. INTRODUCTION

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 [2], testing [3, 4], and debugging [5]. This paper describes the fundamentals of data flow analysis, and specifically dynamic data flow analysis. The paper concludes with a number of requirements for new testing approaches using dynamic data flow analysis. Osterweil and Fosdick [6] 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 undefinition.
2. A definition must be followed by a reference, before another definition or undefinition.

In this context, a reference refers to the use of a variable, definition refers to the assignment, and undefinition 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 is not seen as competing, but as complementary strategies [7].

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 undefined 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*.

II. RELATED WORK

Dynamic data flow analysis has been performed for various procedural [9, 10, 8] and OO [11, 12] 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 [5], and Chen and Low [12] 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.* [11] 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 [13] including source code instrumentation, byte code instrumentation, and instrumentation using the Java Platform Debugger Architecture [14].

III. OVERVIEW OF DYNAMIC DATA FLOW ANALYSIS

Dynamic data flow analysis is a method of analyzing the sequence of actions on data in a program as it is being run. Fosdick and Osterweil [15] mention that there are three types of actions that can be performed on a data item, namely, define (d), reference (r) and undefine (u). A variable is said to be defined if a value is assigned to it; referenced if the value is fetched from the memory; and undefined if the value becomes unknown or inaccessible. During program execution, a variable can be in one of the following four states: state D (defined), state R (referenced), state U (undefined), and state A (abnormal). Huang [8] introduced tracing the data flow anomalies through state transitions instead of sequence of actions. To detect data flow anomalies dynamically, Huang [8] introduced program instrumentation, which is achieved by inserting software probes into the original source program to collect information during the program execution. Different from other methodologies, A. Cain, T.Y. Chen[16] proposed an OO approach to dynamic data flow analysis, which provides a more straightforward and natural solution for OO programs. The main component of this 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 [11, 9, 10, 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. Modeling 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 metamodel 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 metamodel is informed of the destruction of x, the corresponding Variable object is removed from the model.

IV. EXTENSION OF DYNAMIC DATA FLOW ANALYSIS

A.S. Boujarwah[11] presented the dynamic data flow analysis for JAVA programs. He proposed, Unlike C++, Java programs do not offer a way to find the address or the size of a data in a Java program. Thus, the data hiding problem cannot be solved by tracking the use of absolute memory locations associated with the data in Java programs. Instead, we have to use the explicit data name itself. He has proposed the dynamic data flow analysis method for Java programs.

1. Instrumented information

Since each of the usage types of Java variables has different semantics, we focus on each of them alone. Moreover, we consider some variables and operations that should be treated in a special way as follows.

1.1. Local variables

A local variable can be declared within a method body or can exist as a method's formal arguments. In addition, it can be of a primitive or a reference type. The reference type variable can be either an object, or array or string (special kinds of objects).

To differentiate one variable from another, we have to consider its name and location. The location of a local variable contains the class where the method is declared and the method where the variable is declared. Since we may have overloaded methods, we have to consider the method name and the argument types. No two local variables at the same location can have the same name with different types. Thus, considering the variable type is not necessary. For example, in the following code:

```
class A {
void m(int i) { int j=0; int k = 1; }
void m() { int j=2; }
void n(int i) { int j= 1; }
}
class B { void m() { int j= 3; } }
```

For the definition of variable j in m (int i), if the variable name is not considered in the inserted probe, it will be ambiguous with the definition of variable k in the same method. Similarly, the ambiguity occurs with variables j in n(), j in m() in class A, and j in m() in class B, if the method name, method argument types, and class name, respectively, are not included in the inserted probe. For a local object variable, we have to consider each of its attributes separately

1.2. Instance variables

Instance variables are dealt with as attributes of objects declared as local variables. For such attributes, we have to consider two arguments in addition to the attribute name and location (at which the object is declared). The first is the object name; the second is the class or the interface at which the attribute is declared. We will refer to this class as the actioned class/interface. The actioned class/interface can be the object type or one of its superclasses or interfaces. In the following example:

```
class A { int i; void m(int i) { A a= new A(); a.i = i; }
class B extends A{
int i, j;
void m(int i) { B a = new B(); B b=new B(); a.i = i; }
void m() { B a= new B(); a.i= 1; }
void n(int i) { B a = new B(); a.i = i; }
}
```

For the definition of variable i in B.m(int i), if the variable name is not considered, it will be ambiguous with the definition of the instance variable j in class B. Similarly, the ambiguity occurs with instance variables i in n(int i), i in m(), and i in m(int i) in class A, attribute i for object b, and overridden attribute i declared in class A, if the method name, method arguments, class name, object name, and actioned class name, respectively, are not included in the inserted probe. Also, we can consider the returned variables of the methods as instance variables. In this case, we do not have a variable name. To identify this variable, we should consider the method name, its arguments, and the class at which it is declared (actioned class). Moreover, we have to consider the object name - a variable which is considered as one of the object's attributes- and the object location.

```
class A {
int m() { …. }
int m(int j) { …. }
int r() { ….. }
void n() { A a1= new A(); A a2 = new A(); int
j =a1.m(); ….. }
}
class B {
int m() { …. }
void n() { A a1= new A(); …. }
}
```

If, in the reference of returned variable by method m() in the calling statement j= a1.m(), the method name is not considered, it will be ambiguous with the returned value of method r. Similarly, if the method arguments, the actioned class, and the object name and location, respectively, are not considered in the inserted probe, the ambiguity occurs with

the returned variable of method m(int j) in A, the returned variable of method m() in B, the returned A variable from m() in A for object a2 in A.n(), and the returned variable from m() in B for object a1 in B.n(),.

1.3 Class variables

Class variables are global for all objects of the class or the interface at which the variables are declared. Therefore, to identify a class variable we only need to know its name and the class or interface at which it is declared (actioned class/ interface). Consider the following example:

```
class A {
static int j;
static int k;
void m(int i) { A a1= new A(); A a2 =new A();
a1.j = 1; }
}
class B { static int j; }
```

For the definition of variable a1.j in m(), if the variable name is not included in the inserted probe, it will be ambiguous with the instance variable k. Similar ambiguity occurs with class variable j in class B if the actioned class is not determined. Adding the object name and the method name and arguments is not necessary, because the variable is global (i.e. variables a1.j and a2.j share the same memory location).

1.4. Class/interface fields

Class/interface fields are members of class/interface. When a Java program is executed, these fields occupy part of the memory. When an object is created, the defined fields are referenced to define the corresponding attributes of the object. As with identification of class variables, identification of a class/interface field is achieved through an inserted probe that contains the field name and the class/interface name.

1.5. Reference variables

When a variable with a reference type is assigned to another variable, the two variables share the same memory location. Therefore, instead of inserting probes that indicate definition and reference actions included in the assignment statement, we can instrument the following probe: (variable1 = variable2).

V. CONCLUSIONS AND FUTURE DIRECTIONS

This paper has presented an analysis on 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. 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. The general applicability of the meta-meta-model needs to be evaluated by defining a metamodel for a language other than Java. The .NET framework 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.

REFERENCES

[1] F. E. Allen and J. Cocke. A program data flow analysis procedure. Communications of the ACM, 19(3):137–147, 1976.

[2] S. Graham, P. Kessler, and M. McKusick. An Execution Profiler for Modular Programs. Software—Practice and Experience, 13:671–685, Aug. 1983.

[3] S. Rapps and E. Weyuker. Data flow analysis for test data selection. In Proceedings of the Sixth Internaltional Conference on Software Engineering, pages 272–278, 1982.

[4] S. Rapps and E.Weyuker. Selecting Software test data using data flow information. IEEE Transactions on Software Engineering, 11(4):367–375, 1985.

[5] D. A. Price. Program Instrumentation for the Detection of Software Anomalies. Master's thesis, Department of Computer Science, University of Melbourne, Australia, 1985.

[6] L. J. Osterweil and L. D. Fosdick. DAVE–A Validation Error Detection and Documentation System for Fortran Programs. Software Practice and Experience, 6:473–486, 1976.

[7] T. Y. Chen and P. C. Poole. Dynamic dataflow analysis. Information and Software Technology, 30(8):497–505, Oct. 1988.

[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] 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.

[10] 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.

[11] 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.

[12] T. Y. Chen and C. K. Low. Error Detection in C++ through Dynamic Data Flow Analysis. Software – Concepts and Tools, 18:1–13, 1997.

[13] 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 Stateof- the-Art inRuntime Inspection (ASARTI 2003), 2003.

[14] S. Microsystems. Java Platform Debugger Architecture. Available at http://java.sun.com/j2se/1.4.1/docs/guide/jpda/.

[15] L.D. Fosdick, L.J. Osterweil, Data flow analysis in software reliability, Computing Surveys 8 (3) (1976) 305±330.

[16] A. Cain, T.Y. Chen, D.D. Grant, F.-C. Kuo ∗ and J.-G. Schneider," An Object Oriented Approach towards Dynamic Data Flow Analysis", 1550-6002/08 $25.00 © 2008 IEEE DOI 10.1109/QSIC.2008.18

