

Dynamic Data Flow Testing In Object Oriented Paradigm

Mrs. Rashmi*, Mrs. Anju Saha**

University School Of Information technology, Guru Gobind Singh Inderprastha University, Delhi
*rashmibehal@gmail.com, **anju_kochhar@yahoo.com

Abstract-A Large part of computer programs consists of data declarations. Thus, an increased focus on Data Flow Testing should be considered. Data flow Analysis has been applied in testing program by both ways statically and dynamically. Object Oriented program use the concept of Inheritance and Run-Time Polymorphism. So, there is difficulty of finding data points through objects. To achieve this I need to track my objects, which will use the methodology of Meta model and object table. Probing the data positions using the assertion will create object tables. My study will try to achieve the valid and invalid pairs of events.

I. INTRODUCTION

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. 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 study 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.

A. What Is data flow Testing: Data flow testing refers to forms of structural testing which focus on the points at which

variables receive value and the points at which points these values are used.

Definition

Node $n \in G(p)$ is a defining node of the variable $v \in V$, written as $DEF(v,n)$, iff the value of the variable v is defined at the statement fragment corresponding to node n . Input statements, assignment statements, loop control statements, and procedural calls are examples of statements that are defining nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables are changed.

Definition

Node $n \in G(p)$ is a usage node of the variable $v \in V$, written as $USE(v, n)$, iff the value of the variable v is used at the statement fragment corresponding to node n .

Output statements, assignment statements, loop control statements, and procedural calls are examples of statements that are defining nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables remain unchanged.

Definition

A usage node $Use(v, n)$ is a predicate use (denoted as P -use) iff the statement n is predicate statement; otherwise $USE(v, n)$ is a computation use, (denoted as C -use). The nodes corresponding to predicate uses always have an out degree ≥ 2 , nodes corresponding to computation uses always have out degree ≤ 1 .

Definition

A def-clear path with respect to variable v (denoted as d - c path) is a def-use path in which $PATHS(P)$, with initial and final nodes $DEF(v, m)$ and $USE(v, n)$ such that no other node in the path is a defining node of v . The D - U paths that are not def-clear are potential trouble spots.[1]

B. DATA flow Anomalies:

It represents the patterns of data usage which may lead to incorrect execution of the code. The following def. shows the list of anomalies

$\sim d$: first define	it is allowed.
$\sim u$: first use	Potential bug. Data is used
without	definition.

Ud: use then redefined.	define Allowed. Data is used and then redefined.
Uk: use	kill Allowed.
~k: first kill before definition.	Potential bug. Data is killed before definition.
kd kill – define	Allowed. Data is killed and then re-defined.
uu use – use	Allowed. Normal case.
kk kill – kill	Potential bug.
d~ define last	Potential bug.
u~ use last	Allowed.
k~ kill last	Allowed. Normal case.

II. STATIC VS DYNAMIC ANOMALY DETECTION

Data flow testing is based on module’s control flow, it assumes that control flow is correct. Data flow testing has to choose enough test cases so that

- Every “define” is traced to each of its “uses”.
- Every “use ” is traced to corresponding to each of its “define”.

Static Analysis is analysis done on source code without actually executing it.

- E.g., Syntax errors are caught by static analysis.

Dynamic Analysis is analysis done as a program is executing and is based on intermediate values that result from the program’s execution.

- E.g., A division by 0 error is caught by dynamic analysis.
- E.g., Object references, passing argument in objects.

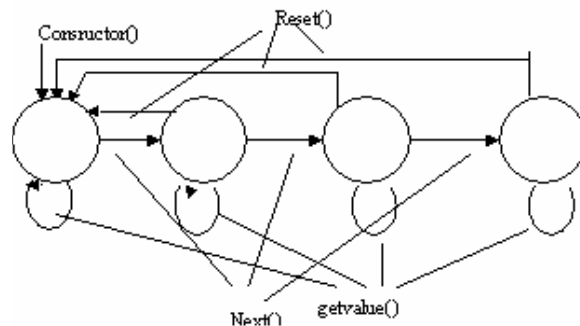
A.. Why Static Data-Flow Testing is not enough?

Static Data-flow Testing will fail in situations where the state of data variable cannot be determined by just analyzing the code. This is possible when the data variable is used as an index for a collection of data elements. for ex. In case of arrays, the index might be generated dynamically during the execution hence we can’t guarantee what the state of the array element is which is referenced by that index. Moreover, the static data flow testing might denote a certain piece of code to be anomalous which is never executed and hence not completely anomalous.[2]

B. Anomalies are Resolved By Finite State Automata

FSAs are objects whose next state **is** determined by the next feature that is called and the current state. The next feature called in conjunction with its parameter values are known **as** stimuli. state (error).

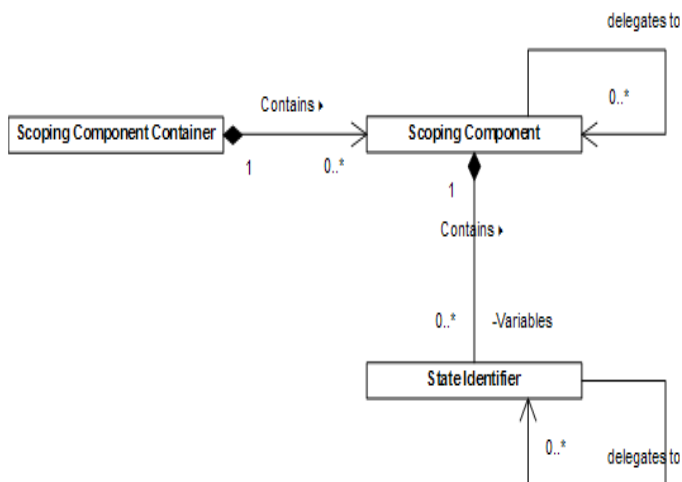
- It can change the object's state to the appropriate new state.
- It can leave the object's state as it is.
- It can change the object's state to an undefined



C. The state diagram of an example Deterministic Finite State Automata

As can be seen in the above figure, the FSA has four states corresponding to the numbers one through to four. There are four different features available for a client to call (including a constructor) and their effects are shown above. The feature Constructor() initializes the FSA to its starting state - one. The feature Next() causes the FSA to change to the next state in the sequence. The feature GetValue() does not affect the current state, it simply informs the client (caller) what the current value is. The Reset() feature allows the state to be returned to the start (one) at any time. The counting FSA would be relatively trivial to test, especially as all features are defined over the whole range of states with no exceptions. Each feature would be called with the object in each of the four states. The result of the calls would be verified against the diagram above. Any feature has five types of possible responses to a particular state and stimuli combination (see figure[3]).

- It can change the object's state to the appropriate new state.
- It can leave the object's state as it is.
- It can change the object's state to an undefined state (error).
- It can change the object's state to a defined, but inappropriate state (error).
- It can leave the object's state **as** it is when it is supposed to change it (error).



III. ADDITIONS TO THE CLASS

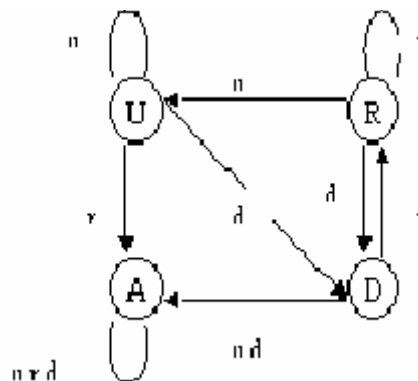
A major part of dynamic data flow testing is the determining of the object's current state. To enable this, a new version of the class under test must be produced with at least one new feature per substate. These enable the tester to inspect the value of chosen substates. However, if there is a state change between two values which were both considered to be part of the same general group, then the change is undetectable. This problem is easily rectified by the addition of an extra set of data-members, whose purpose is to mirror the value of the original data members that the user is interested in. For each data-member being mirrored, an extra feature is required to test the difference between the original and the mirroring data-member, and to update the value of the mirror. As a simple recommendation, it is advisable to insert statements at the beginning and end of each feature to report to the screen (or to a file) the value of the parameters passed and the values returned by the features. It is useful if this is also performed for the substate testing features by tracing the execution of the test case as it provides a useful aid in debugging any errors that might occur. It is usual that some features of a class must perform more than one task or activity to achieve their desired functionality. The greater the number of tasks, the more difficult they will be to test and debug, if errors are present. As a suggestion to aid the tester, it is advisable to insert *assertions* (statements about the current state of the data) into the code between each task, checking that each sub-task as well as the whole task was satisfactorily performed. it involves the creation of states for not only the main object under test, but also the objects that will be passed as parameters.

V. DATA FLOW ANALYSIS

Data flow analysis was originally used as a technique for code optimization in compilers.

[4]It has also been shown to be a useful technique in other areas, such as performance tuning[5], testing[6,7], and debugging[8]. This study describes the fundamentals of data flow analysis, and specifically dynamic data flow analysis. The study concludes with a number of requirements for new testing approaches using dynamic data flow analysis.

Dynamic data flow analysis is a method for analyzing the sequence of actions on data in a program as it is being run. Huang introduced tracing the data flow anomalies through state transitions instead of sequences of actions. When an action is applied on a variable, its state follows transitions according to the state transition diagram shown in Fig. A variable entering an abnormal state indicates a data flow anomaly. Thus, there are three anomalous paths PDUP, PDDP, and PURP, where P is an arbitrary path expression. d,r,u [12]



STATE TRANSITION DIAGRAM

To detect data flow anomalies dynamically, we insert software probes into the original source program to gather information during the program execution.

VI. 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.[10] used a combination of string identifiers to track actions upon variables..

Object oriented are difficult because of Inheritance and run-time binding. Here, we need to trace objects carefully and maintain object tables with their locations so this can be used by other name which can be called as meta process(object of

object). To solve the problem of runtime execution of object-oriented program so we can use meta model.

Meta Model

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 [11] 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. At least, 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 program
4. The output generated by the approach must enable programmers to identify the location and type of any anomalies produced.
5. 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# have two different kinds of types. These are referred to as primitive and reference types in the Java language specification, and as value and reference types in the CLI standard. Variables of a reference type contain a reference to an object or bit sequence. 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. 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.

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. 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.

VII. 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. Secondly, data flow analysis approaches provide a mechanism that allows action information to be extracted 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 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 .

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.

A. Extension of dynamic data flow analysis

Here I, explain the information needed to be included in the inserted probes (these are locations which shows where the object is). Then, I discuss the locations at which the probes should be inserted. Finally, we discuss the dynamic data flow analysis method for object oriented programs. The usages of variables have different semantics. Each is treated separately as follows.

Local Variables: A local variable can be declared within a method body, it can be one of its arguments, or it can be of a primitive or a reference type. To differentiate a local primitive or array type variable from another, we have to consider its

name and location. The location of a local variable contains the class at which the method is declared and the method at which the variable is declared.

Instance Variables: Instance variables are dealt with as attributes of objects declared as local variables. We have to consider the object name and its type (class or interface) besides considering the attribute name and its location (at which the object is declared), and the class or the interface at which the attribute is declared. We can also consider the returned variables of the methods as instance variables.

Class Variables: Class variables are global for all objects of the class or the interface at which the variables are declared.

Reference Variables: When a variable with a reference type is assigned to another, the two variables share the same memory location (i.e., aliasing).

B. Instrumentation

Lastly Boujarwah et al[13]. 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. 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.

C. 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 object 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.

Example: Bill Calculation

Billing Rules	
Usage(min)	Bill(Rs.)
<100	40.0
101-200	50 paise for every additional minute.
>200	10 paise for every additional minute

The source code for above application is:

```

Class bill
{
public:
    int usage;
    float call_duration,pulse_rate;
}

public static int calculate_call(float call_duration,float pulse_rate)
{
    usage=call_duration/pulse_rate;
    return usage;
}

public static double calculatebill(int usage)
{
}

public static double calculateBill(int Usage)
{
    double Bill = 0;
    if(Usage > 0)
    {
        Bill = 40;
    }
    if(Usage > 100)
    {
        if(Usage <= 200)
        {
            Bill = Bill + (Usage - 100) * 0.5;
        }
        else
        {
            Bill = Bill + 50 + (Usage - 200) * 0.1;
        }
        if(Bill >= 100)
        {
            Bill = Bill * 0.9;
        }
    }
}
    
```

bill is a variable which is being calculated at run time as depends upon usage.so object has to track both the methods and maintain its data and use further.

CONCLUSION

In this paper, I proposed a meta model for objects in object-oriented programs. This technique is a specification-based testing using FSMs and by inserting probes. By using FSMs, we can properly model the interactions between data members and member functions. In order to test the behavior of classes effectively, i have used the concept of probes which

are needed to trace objects. Object tables are maintained with their locations so this can be used by other name which can be called as meta process (object of object). To solve the problem of runtime execution of object-oriented program so we can use meta model.

When we analyze a dynamic unit namely an individual unit of that class, however we may find that every pair of events can only be “always valid” or never valid.

transaction on Software Engineering, Vol. 5, 1979, pp. 226-236.

13. T. Y. Chen, H. Kao, M.S. Luk, and W.C. Ying, 'COD-a dynamic data flow analysis system for Cobol', *Information and Management*, Vol. 12, Feb 1987, pp. 65-72.

REFERENCES:

1. Software Testing Book: A Craftsman Approach, second edition. Published at CMC Press.
2. Ref boris beizer, "software testing techniques", international Thomson computer press, 1990.
3. Myers, G. J., *The Art of Software Testing*, John Wiley, 1979
4. F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137-147, 1976.
5. S. Graham, P. Kessler, and M. McKusick. An Execution Profiler for Modular Programs. *Software—Practice and Experience*, 13:671-685, Aug. 1983.
6. S. Rapps and E. Weyuker. Data flow analysis for test data selection. In *proceedings International Conference on Software Engineering*, pages 272-278, 1982.
7. S. Rapps and E. Weyuker. Selecting Software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367-375, 1985.
8. D. A. Price. Program Instrumentation for the Detection of Software Anomalies. Master's thesis, Department of Computer Science, University of Melbourne, Australia, 1985.
9. T. Y. Chen and C. K. Low. Dynamic Data Flow Analysis for C++. In *Proceedings of the Second Asia Pacific Software Engineering Conference*, pages 22-28. IEEE Computer Society, 1995
10. 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.
11. 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)*, July 2003.
12. J.C. Huang, 'Detection of data flow anomaly through program instrumentation', *IEEE*