ISSN: 0975-766X
CODEN: IJPTFI
Available Online through *Research Article*
www.ijptonline.com

DATA-FLOW ANOMALY ELIMINATION USING INSTRUMENTATION WITH THE HELP OF COMPILER

M. Chandru, S. Vijayakumar, Dr. B. Jayanthi
Asst. Professor in Computer Science, Kongu Arts and Science College (Autonomous), Erode, Tamil Nadu, India.
Asst. Professor in Computer Science (P.G), Kongu Arts and Science College (Autonomous), Erode, Tamil Nadu, India.
Assoc. Professor in Computer Science (P.G), Kongu Arts and Science College (Autonomous), Erode, Tamil Nadu, India.
Email: emchandru@gmail.com

Received on: 15.10.2016 *Accepted on: 12.11.2016*

Abstract

Software testing is an important activity that encompasses the whole development and maintenance process of a software application. The objective of testing is to find defects in data object specifications, design artifacts and implementation. Software testing techniques can be divided into 2 kinds: black box and white box techniques. Data-flow testing is a white box testing technique that can be used to detect improper use of data by examining the lifecycle of data variables. However, to date, limited work has been done to monitor data object states to prevent faults.

The main goal of this paper is to discuss the concept of data-flow testing and apply it to a running example. This work deals with incorporating instrumentation in data object states to remove the unused data objects so as to avoid the data flow anomalies such as defined but not used for anything (DK) with help of the compiler. It also used to manage memory as well.

Keywords: Data-flow testing, Data Objects, Instrumentation, Data-flow anomalies.

Introduction

Software testing is an expensive component of the software development and maintenance process. Since testing can be manual, automated or the combination of both. When it comes to type of testing, there are two types: Static testing and Dynamic testing. In static testing the actual source code analyzed without executing it. While in dynamic code is executed to find all possible bugs. The main goals of software testing are to reveal bugs and to ensure that the system being developed complies with the customer's requirements. To make testing effective, it is recommended that test planning / development begin at the onset of the project. Software testing techniques can be divided into 2 kinds:

Black box techniques and White box techniques Black box testing is mainly a validation technique that checks to see if the product meets the customer requirements. However, white box testing is a verification technique which uses the source code to guide the selection of test data. Data-flow testing is a white box testing technique that can be used to detect improper use of data values due to coding errors. Errors are in advertently introduced in a program by programmers. For instance, a software programmer might use a variable without defining it. Additionally, he/she may define a variable, but not initialize it and then use that variable in a predicate [1].

Data-flow testing focuses on the variables used within a program. An overwhelming majority of programs written today handle data. Most programming language paradigms utilize the concept of variables: marked sections of memory that can be assigned (and reassigned) a particular value for example, an integer or ASCII character. Multiple variables can be used together to calculate the values of other variables; and variables can receive their values from other sources – such as human input via a keyboard, for instance. The concept of Data-Flow testing allows the tester to examine variables throughout the program, helping him/her to ensure that none of the aforementioned errors occur. Data -flow testing is a refinement of control flow testing technique. Data flow testing has no relation with data flow diagram. It examine the lifecycle of a data object i.e. its definition, its usage in the program, its computation and killing of that data object. When the code is examined with respect to defined data objects in the program, one must examine where the usage of that data object occurs in that program. If that data object is been used in the program after it has been killed is an anomaly. In this paper, we have discussed the concept of data-flow testing and data-flow anomalies. The paper is organized as follows Section 2 discusses about related work. Section 3 describes about problem statement. Section 4 discusses about the proposed algorithm. Section5 discuss about experimental analysis. Section 6 Concludes the paper with future work.

Related Work

Traditionally there are two methods of testing software: Black Box Testing and White Box Testing. White Box testing techniques are Condition testing, Basis path testing, Flow Graphs, Loop testing and Data flow testing. Data flow testing selects test paths of a program based on definition and use of variables in the program. Statements in a program are related to each other according to definitions and uses of variables. The work discussed in this paper relates to the previous work on data flow testing and analysis. Data flow analysis has been investigated since the late sixties in many different contexts, starting from program optimization [2] and computer architectures and proposed for application to software testing since the mid seventies. Researchers tackled the evaluation of the effectiveness of

data flow testing from a different angle, comparing between data flow and mutation testing. Experiments indicate that satisfying mutation testing generally requires more test cases than satisfying data flow testing, but data flow testing is as much effective and thus preferable because easier to satisfy. The author Frankl, [2] uses all-uses criteria in her work "An Applicable Family of Data Flow Testing Criteria". This was stronger criteria. It focuses on all p-uses and c-uses of each and every variable hence covering each and every path and branch of software under the test. The contextual def-use associations introduced by Souter and Pollock [3] led to data flow testing approaches that capture the structural characteristics of object oriented designs. The most common approach to monitoring for data-flow testing is a technique known as last definitions. In this approach Jonathan Misurda, [4] describe a new scalable and flexible framework for testing programs with a novel demand-driven approach based on execution paths to implement test coverage. This technique uses dynamic instrumentation on the binary code that can be inserted and removed on-the-fly to keep performance and memory overheads low. The author Ira R. Forman, [5] An algebra A is developed that is specialized for the detection of data flow anomalies by interpreting the regular expression for the paths in a program as an A expression. Two methods are subsequently presented that use A but do not require the explicit computation of the regular expression for the paths. One method is based on the prime program decomposition. The other is based upon the iterative algorithms of global data flow analysis. In addition the use of the algebra to get better warning messages than just the detection of anomalies is presented. Mark New, in his work describes, [6] a form of structural (white box) testing that is a variant on path testing, focusing on the definition and usage of variables, rather than the structure of the program. The author Prof. Y. N. Srikant [7] says about in compiler design, which a variable x is live at a point p, and then you know the value of that particular variable at that point is going to be used along some path in the flow graph, starting at p. If this does not happen, then say that x is dead at p. The author BG Geetha, [8] says that by using all the used paths, the errors such as "variable defined but not used in the program" can be detected. In this paper, we have discussed the use of data-flow testing to determine Variables defined in the program may not be referenced throughout the program.

Problem Statement

This paper discusses the data flow testing concepts, anomalies and related definitions. Data flow testing is a structural testing approach and is basically a verification technique which uses the source program to find the anomalies, ie., unused data objects. Flow graphs are used as a basis for dataflow testing as in the case of path testing. Variables used in the program may be defined and referenced throughout the program. There may be few unused/referenced

anomalies in the program. This paper highly focuses on unused/referenced anomaly that is "A variable is defined but not used anywhere".

A. *Definitions*

Consider a program P with a flow graph F, initial node n, final node m and a set of program variables V.

i. DEF (v1, v2, n): A node n of flow graph F where the variables v1,v2 Є V is defined.

ii. USE (v1, n): A node n of flow graph F where the variable v1 Є V is used.

iii. NUSE (v2, n): A node n of flow graph F where the variable v2 Є V is not used

iv. DU Path (definition use): A path in a flow graph F where at the initial node n of the path a variables v1,v2 Є V is defined and at the final node m of the path the variable v1 Є V is used/referenced, where as v2 Є V is unused/not referenced.

v. DC Path (definition clear): A DC path of the flow graph F in which no node between the initial node n and final node m of the path is a USE (v1, n) and NUSE(v2,n).

B. *Data Flow Anomalies*

Data-flow testing uses the control flow graph to explore the unreasonable things that can happen to data objects (i.e., anomalies[9]).

i. A variable is defined but not used.

ii. A variable is used but never defined.

iii. A variable is defined twice before it is used.

An object (e.g., variable) is defined when it:–appears in a data declaration–is assigned a new value–is a file that has been opened–is dynamically allocated, ect,. We can note this by a letter d. An object is killed when it is:– released (e.g., free) or otherwise made unavailable (e.g., out of scope)–a loop control variable when the loop exits–a file that has been closed, ect,. We denote this by a letter k. An object is used (denoted by a letter u in common) when it is part of a computation or a predicate. An anomaly is denoted by a two-character sequence of actions. For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage. What is an anomaly is depend on the application. There are nine possible [10] two-letter combinations for d, k and u. some are bugs, some are suspicious, and some are okay.

1. **dd** :- probably harmless but suspicious. Why define the object twice without an intervening usage?
2. **dk** :- probably a bug. Why define the object without using it?

3. **du** :- the normal case. The object is defined and then used.
4. **kd** :- normal situation. An object is killed and then redefined.
5. **kk** :- harmless but probably buggy. Did you want to be sure it was really killed?
6. **ku** :- a bug. The object does not exist.
7. **ud** :- usually not a bug because the language permits reassignment at almost any time.
8. **uk** :- normal situation.
9. **uu** :- normal situation.

In addition to the two letter situations, there are six single letter situations. The use of a leading dash mean that nothing of interest (d,k,u) occurs prior to the action noted along the entry-exit path of interest. A trailing dash to mean that nothing happens after the point of interest to the exit.

The possible anomalies are:

10. **-k** :- possibly anomalous because from the entrance to this point on the path, the variable had not been defined. Killing a variable that does not exist.
11. **-d** :- okay. This is just the first definition along this path.
12. **-u** :- possibly anomalous. Not anomalous if the variable is global and has been previously defined.
13. **k-** :- not anomalous. The last thing done on this path was to kill the variable.
14. **d-** :- possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
15. **u-** :- not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.

C. Data Flow Anomaly State Graph

Data flow anomaly model prescribes that an object can be in one of four distinct states:

1. **K** :- undefined, previously killed, does not exist
2. **D** :- defined but not yet used for anything
3. **U** :- has been used for computation or in predicate
4. **A** :- anomalous

These capital letters (K, D, U, A) denote the state of the data object and the programs' action denoted by lower case letters.

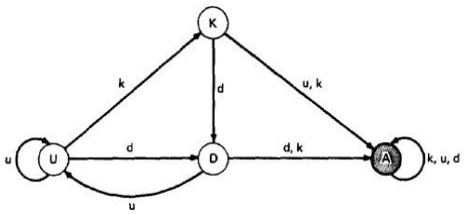

Figure 1: Data Flow Anomaly State Graph.

Assume that in figure 1, the data object starts in the K state - that is, it has not been defined or does not exist. If an attempt is made to use it or to kill it, the object's state becomes anomalous (state A) and, once it is anomalous, no action can return the data object to a working state. If it is defined (d), it goes into the D, or defined but not yet used, state. If it has been defined (D) and redefined (d) or killed without use (k), it becomes anomalous, while usage (u) brings it to the U state. If in U, redefinition (d) brings it to D, u keeps it in U, and k kills it.

Data-flow testing strategies differ in the extent to which predicate uses and/or computational uses of variables are included in the test set.

All Definitions Strategy (AD) : The all definitions strategy asks only every definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use.

There are various types of data flow testing strategies; this paper focuses on **All Definitions Strategy (AD).** The all definitions strategy asks only every definition of every data object is covered by at least one use of that data object.

Proposed Algorithm-Instrumentation Algorithm

The proposed instrumentation algorithm consists of the following stages. The steps (2) to (4) are repeatedly done as long as DK exists.

1. Step 1: Read the compiler warning output which enlist the line number of unused data objects (DK).
2. Step 2: Identify the DK in the source file with respect to line number.
3. Step 3: Read the definition of the DK in source file.
4. Step 4: Eliminate the definition of DK.
5. Step 5: Recompile the source program.

The instrumentation algorithm reads the compiler message in specific format like line number, variable / data object and its state (Unused, dead and not initialized etc.) This algorithm get the line number and unused data object from the compiler message file/window, than matches with source code, reads the entire line with the help of language

delimiter. After identifying the particular data object and its definition the algorithm eliminates the line or comment the line to prevent reading the line by the compiler. The compiler warns the definitions, uses and kills of data objects depending on the type of D, U, K, A. The instrumentation program is developed in C language, which reads the compiler warnings from a text file. The instrumentation program reads all the unused definitions of data objects one by one and removes from the source file by pointing the line numbers. The experiments are carried out in a .NET application with unused variable and data object definitions. By using the Microsoft Management Consoles' Performance Logs and Alerts supports which monitors the utilization of operating system resources in detail, the metrics was made in terms of memory. This technique is applicable to all source codes which is readable by C program and have formatted compiler message file.

Discussion on Experimental Analysis

The experimental analysis was carried out with the different modules of the application and data objects like simple variables, ASP.Net objects, Namespaces and third-party control objects. The proposed algorithm is tested against a mid-sized student's attendance management application, developed using C# and ASP.Net. This technique is applicable to all source codes which is readable by C program and have formatted compiler message file.

The empirical results confirm the complexity results and suggest that the technique proposed in this paper work scales up well to mid-size programs and it performed well in few minutes. The result shows the improvement in terms of memory and time is direct proposal to the number of objects and its type. The test dataset i.e., students attendance management application, contains all the types of data objects and is portable to the compiler to generate its state like defined, used, unused, dead etc., The experimental analysis was carried out with the different modules of the application and data objects like simple variables, ASP.Net objects, Namespaces and third-party control objects. The result shows the improvement in terms of memory and time is direct proposal to the number of objects and its type.The following figure no.2 shows the compiler message where the variable is assigned and is never used.

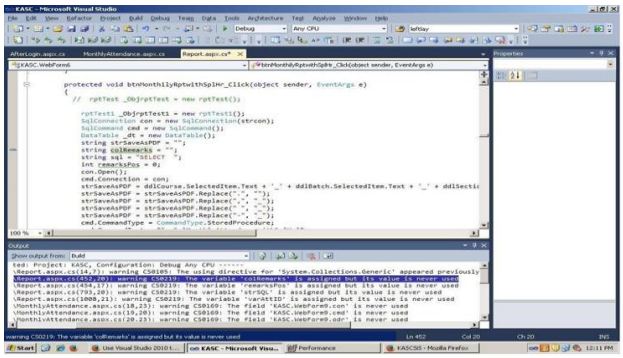

Figure No: 2

The following figure no.3 shows the performance evaluation of the source code with unused data object in terms of memory.

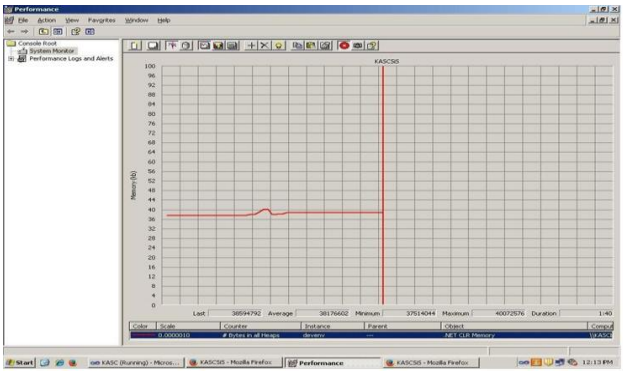

Figure No: 3

The following figure no.4 shows the eliminated data objects which are never used in the source code.

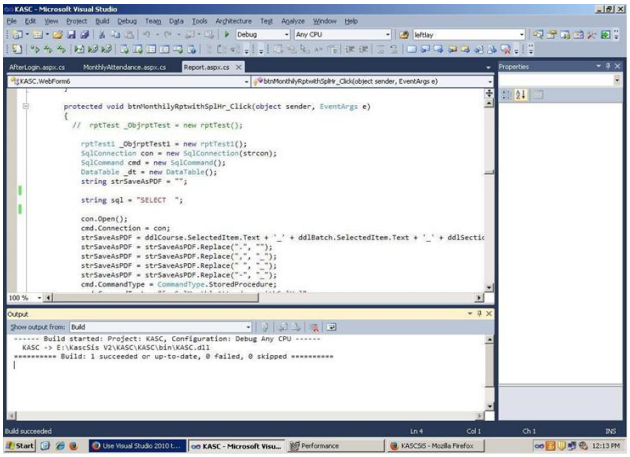

Figure No: 4

The following figure no.5 shows the performance evaluation of the source code after eliminating the unused data objects.

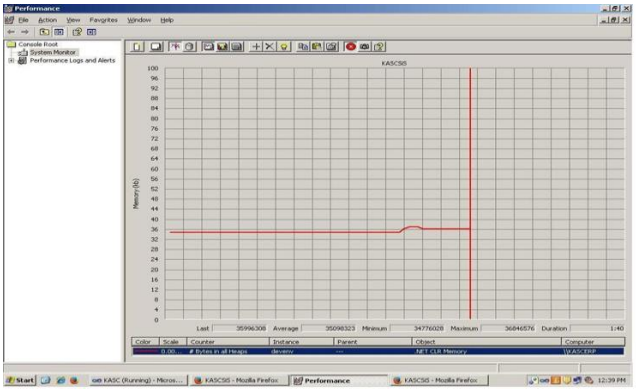

Figure No: 5

The following figure no.6 shows the compiler message where the different data objects are assigned and is never used.

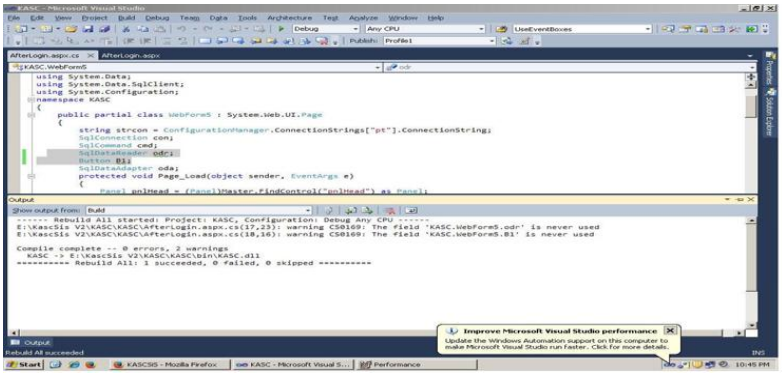

Figure No: 6

The following figure no.7 shows the compiler message where the different data objects with third-party objects are assigned and is never used.

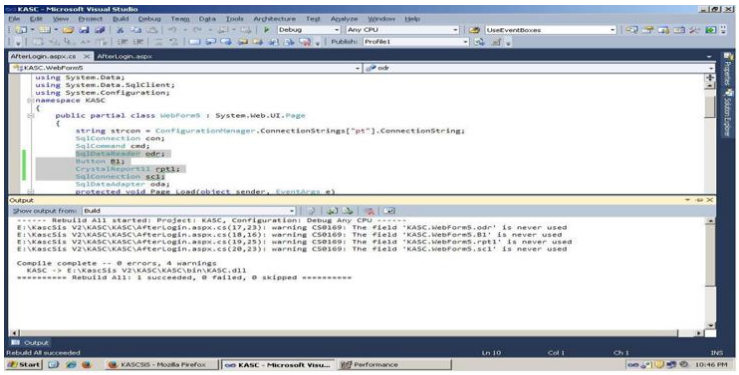

Figure No: 7

The following figure no.8 shows the performance evaluation of the source code for data objects shown in the above figures no.6 and no.7. The figure show the performance evaluation based on number of data objects and its type.

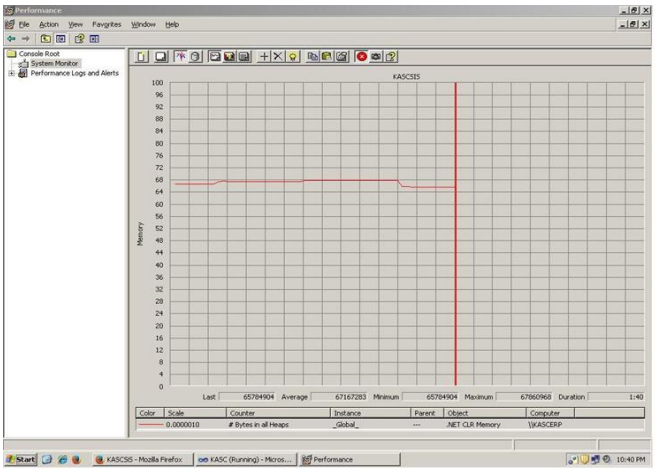

Figure No: 8.

Conclusion & Future Work

The concepts of defined and unused data object testing by its variables allow the tester to examine the program in a different way. Since an overwhelming majority of programs are written using variables, and therefore rely on the

patterns of variable definition, used (as well as the values assigned to the variable) and unused/not referenced, data flow testing can be viewed as a useful addition to the tester's toolbox.

Whilst adopting a data testing techniques may not detect all define/unused data object, the fact that this instrumentation technique can help to detect define/unused data object that may not be obvious during functional testing (or even other forms of structural testing) means that at least some level of data flow testing should be seriously considered during the testing process. In future this technique can be analyzed and experimented with bottom up approach. Dead code elimination can adopted to deploy define-kill with proper experimentation and analysis in future.

Acknowledgment

We would like to express our gratitude to our management for providing support and encouragement to take our research work in this paper.

References

1. Lee Copeland "A Practitioner's Guide to Software Test Design", STQE Publishing, 2004.
2. Phyllis G. Frankl and Elaine J. Weyuker "An Applicable Family of Data Flow Testing Criteria" IEEE Transactions on Software Engineering, vol. 14. No. 10, October 1988, pp 1483-1498.
3. A. L. Souter and L. L. Pollock, "The construction of contextual defuse associations for object-oriented systems," IEEE Transactions on Software Engineering, vol. 29, no. 11, pp. 1005–1018, 2003.
4. Misurda, J., Clause, J., Reed, J., Childers, B. Rr., and Soffa, M. "Demand-driven Structural Testing with Dynamic Instrumentation". In International Conference on Software Engineering (May 2005), pp. 156–165.
5. Forman, I. R. "An Algebra for Data Flow Anomaly Detection". Seventh International Conference on Software Engineering, Orlando, Fl., March 26-29, 1984.
6. Mark New, "Data Flow Testing" - CS-399: Advanced Topics in Computer Science (321917)
7. Prof. Y. N. Srikant "Compiler Design", Department of Computer Science and Automation,IISC Bangalore.
8. B G Geetha, "A Tool for Testing of Inheritance Related Bugs in object oriented programs" – Ph.D., Thesis.-2014.
9. Perry, William E. "Effective Methods for Software Testing" 3rd ed. Wiley Publishing, 2006.
10. Boris Beizer, "Software Testing Techniques", International Thomas Computer Press,1990.