Data Flow Testing

CS-399: Advanced Topics in Computer Science

Mark New (321917)

Abstract

This document discusses data flow testing: a form of structural (white box) testing that is a variant on path testing, focussing on the definition and usage of variables, rather than the structure of the program. Two different topics relating to data flow testing are discussed: define/use testing, along with a set of test coverage metrics; and the concept of splitting a program into slices, according to its variables, to ease the testing of large software systems.

Contents

1 Introduction 2

2 Example Used 4

3 Define/Use Testing 6

4 Program Slices 13

5 Conclusion 16

References 17

1 Introduction

An overwhelming majority of programs written today handle data. Most programming language paradigms utilise 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.

This increased level of complexity can result in errors within programs: references may be made to variables that don't exist, or the value of variables may be changed in an unexpected and undesired manner. The concept of Data Flow Testing allows the tester to examine variables throughout the program, helping him to ensure that none of the aforementioned errors occur.

Data flow testing can be considered to be a form of structural testing: in contrast to functional testing, where the program can be tested without any knowledge of its internal structures, structural testing techniques require the tester to have access to details of the program's structure. Data flow testing focuses on the variables used within a program. Variables are defined and used at different points within the program; data flow testing allows the tester to chart the changing values of variables within the program. It does this by utilising the concept of a program graph: in this respect, it is closely related to path testing, however the paths are selected on variables.

There are two major forms of data flow testing: the first, called define/use testing, uses a number of simple rules and test coverage metrics; the second uses "program slices" – segments of a program.

The importance of analysing the use of variables in programs has been recognised for a long time. Compilers for languages such as COBOL introduced a feature in which

Variables have been seen as the main areas where a program can be tested structurally. Early methods of data testing involved static analysis: the compiler produces a list of lines at which variables are defined or used. The term static analysis refers to the fact that the tester does not have to run the program to analyse it. Static analysis allows the tester, according to Jorgensen, to focus on three "define/reference anomalies" [1]:

> "A variable that is defined but never used (referenced).
> A variable that is used but never defined.
> A variable that is defined twice before it is used."

Static analysis is still used. For instance, a search of the World Wide Web reveals a number of tools that perform static data flow analysis. One tool, AS-

SENT, by Tata Consulting Services, is described as "a global data flow static analysis tool that automatically ensures conformance of C/C++ and Java code"[1]. Another example of a static analysis tool is called LDRA Testbed[2].

Dynamic data testing is different: by selecting, for instance, paths through the program according to the locations and properties of references to variables within the program code, a program can be analysed in terms of how the variables are affected, assigned and changed throughout the course of the program when running with certain test data. This complements (or maybe even replaces) the concept of selecting paths according to the structure of the program (looking at its loops, branches etc.).

An alternative approach is to again look at the program according to its variables; however, this time to 'slice' the program into a number of individually executable components, each focussing on one particular variable at one particular location within the program. It will then be possible to examine the program with respect to those variables without having to examine the entire program. Relationships and linkages between variables can then more easily be examined, and the slices can be tested individually.

This document will provide a relatively informal and concise description of these data flow testing techniques, hopefully providing a sense of their usefulness as part of an overall testing strategy.

[1]Website: http://www.tcs.com/0_products/assent/index.htm; description quoted from http://www.testingfaqs.org/t-static.html#ASSENT

[2]Website: http://www.ldra.co.uk/

2 Example Used

Throughout this document, concepts related to data flow testing will be demonstrated or illustrated using an example. This example will be briefly outlined below.

Staff Discount Program

The following program is used in a hypothetical retail situation. The owner of a shop has decided that her staff can have a 10 percent discount on all their purchases. If they spend more than £15, then the total discount is increased by 50 pence. The price of each item being purchased is input into the program. When -1 is entered, the total price is displayed, as well as the calculated discount and the final price to pay.

For example, the values £5.50, £2.00 and £2.50 are input, equalling £10.00. The total discount would equal £1.00 (10% of £10.00), with the total price to pay equalling £9.00.

A second example would have purchases of £10.50 and £5.00, equalling £15.50. In this case, as the total value is over £15, the discount would be £2.05 (10% of £15.50 is £1.55, plus 50p as the original total is over £15), meaning that the total price to pay would be £13.45.

The source code, written in pseudocode, for a program which has been written to perform the task described above, is shown below:

```
1  program Example()
2  var staffDiscount, totalPrice, finalPrice, discount, price
3  staffDiscount = 0.1
4  totalPrice = 0
5  input(price)
6  while(price != -1) do
7      totalPrice = totalPrice + price
8      input(price)
9  od
10 print("Total price: " + totalPrice)
11 if(totalPrice > 15.00) then
12     discount = (staffDiscount * totalPrice) + 0.50
13 else
14     discount = staffDiscount * totalPrice
15 fi
16 print("Discount: " + discount)
17 finalPrice = totalPrice - discount
```

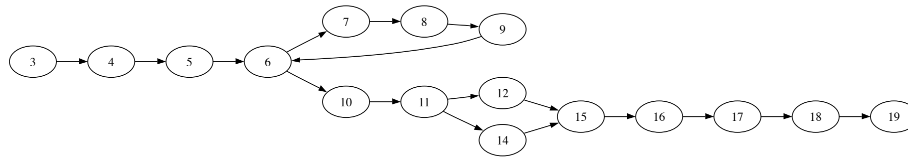

Figure 1: The program graph for the example code

```
18  print("Final price: " + finalPrice)
19  endprogram
```

The program (or control flow) graph for this program is shown in figure 1. Each node in the graph corresponds to a statement in the program[3]; however, lines 1 and 2 do not correspond to any node. This is because these lines are not used in the actual code of the program: they are used by the compiler to indicate the start of the program and to assign space in memory for the variables respectively. Similarly, lines consisting entirely of comments would not be included in the graph.

The while and if-then-else code blocks in the program are represented on the graphs in a clear way. Nodes 6 to 9 correspond to the while loop, and nodes 11 to 15 correspond to the if-then-else block. The iteration of the while look is represented by a loop from 6 to 9; when the price variable equals -1, the flow of control goes from node 6 to 10.

The program graph shows that there is no loop within the if-then-else block: the flow of control can go from node 11 to either node 12 or node 14. This corresponds to the different paths that could be followed within the if-then-else block: either the condition evaluates to true, at which point line 12 is executed, or the condition evaluates to false, so line 14 is executed.

Program graphs allow the tester to view the structure of the program visually. The structure of programs with many control-flow statements (if statements, while loops, etc.), particularly when nested, can be difficult to decipher when viewed in source code form. Generating program graphs allows the tester to make use of certain data flow testing techniques. These will be covered in the next section.

[3]Strictly, the node corresponds to a statement fragment – for instance, in a language such as C, multiple actions can be combined into one statement, using a semi-colon as a delimiter. For example: `function1(); function2(); c = function3();`. In addition, multiple conditions can be used in a control-flow statement. For example: `if(x == 2 || y == 3)`.

3 Define/Use Testing

Define/Use testing uses paths of the program graph, linked to particular nodes of the graph that relate to variables, to generate test cases. The term "Define/Use" refers to the two main aspects of a variable: it is either defined (a value is assigned to it) or used (the value assigned to the variable is used elsewhere – maybe when defining another variable). Define/use testing was first formalised by Sandra Rapps and Elaine Weyuker in the early 1980s [2].

Define/use testing is meant for use with structured programs. The program is referred to as P, and its graph as G(P). The program graph has single entry and exit nodes, and there are no edges from a node to itself. The set of variables within the program is called V, and the set of all the paths within the program graph P(G) is PATHS(P).

Within the context of define/use testing, with respect to variables there are two types of nodes: defining nodes and usage nodes. The nodes are defined as follows:

> **Defining nodes, referred to as DEF(v, n):** Node n in the program graph of P is a defining node of a variable v in the set V if and only if at n, v is defined. For example, with respect to a variable x, nodes containing statements such as "input x" and "x = 2" would both be defining nodes.
>
> **Usage nodes, referred to as USE(v, n):** Node n in the program graph of P is a usage node of a variable v in the set V if and only if at n, v is used. For example, with respect to a variable x, nodes containing statements such as "print x" and "a = 2 + x" would both be usage nodes.

Usage nodes can be split into a number of types, depending on how the variable is used. For instance, a variable may be used when assigning a value to another variable, or it may be used when making a decision that will affect the flow of control of the program.

The two major types of usage node are:

- **P-use:** predicate use – the variable is used when making a decision (e.g. if $b > 6$).
- **C-use:** computation use – the variable is used in a computation (for example, $b = 3 + d$ – with respect to the variable d).

There are also three other types of usage node, which are all, in effect, subclasses of the C-use type:

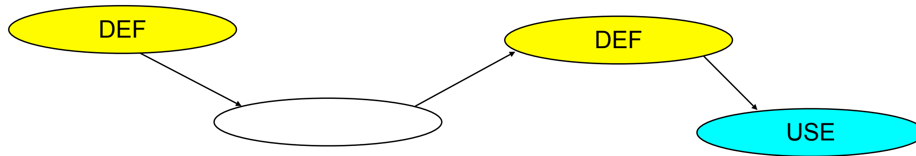

Figure 2: An example of a DU-path

- **O-use:** output use – the value of the variable is output to the external environment (for instance, the screen or a printer).
- **L-use:** location use – the value of the variable is used, for instance, to determine which position of an array is used (e.g. `a[b]`).
- **I-use:** iteration use – the value of the variable is used to control the number of iterations made by a loop (for example: `for (int i = 0; i <= 10; i++)`).

Looking at the example covered in section 2, for the variable `totalPrice` table 1 lists the defining and usage nodes (the particular instance of the variable being referred to is highlighted in **bold text**).

Node	Type	Code
4	DEF	**totalPrice** = 0
7	DEF	**totalPrice** = totalPrice + price
7	USE	totalPrice = **totalPrice** + price
10	USE	print("Total price: " + **totalPrice**)
11	USE	if(**totalPrice** > 15.00) then
12	USE	discount = (staffDiscount * **totalPrice**) + 0.50
14	USE	discount = staffDiscount * **totalPrice**
17	USE	finalPrice = **totalPrice** - discount

Table 1: The defining and usage nodes for the variable totalPrice

With these nodes, some useful paths can be generated.

Definition-use (du) paths: A path in the set of all paths in P(G) is a du-path for some variable v (in the set V of all variables in the program) if and only if there exist DEF(v, m) and USE(v, n) nodes such that m is the first node of the path, and n is the last node.

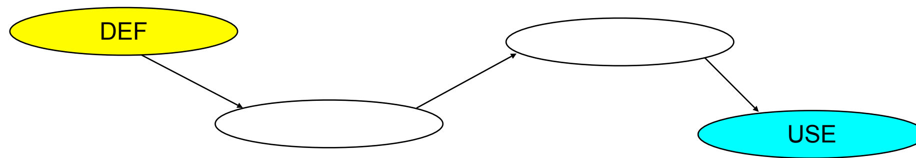

Figure 3: An example of a DU-path that is also definition-clear

> **Definition-clear (dc) paths:** A path in the set of all paths in P(G) is a dc-path for some variable v (in the set V of all variables in the program) if and only if it is a du-path and the initial node of the path is the **only** defining node of v in the path.

Figure 2 shows an example of a du-path for a variable x in a hypothetical program. However, this path is not definition-clear, as there is a second defining node within the path. Figure 3, on the other hand, is definition-clear.

Looking at the example in section 2, for the price variable there are two defining nodes and two usage nodes, as listed below:

- Defining nodes:
  - DEF(price, 5)
  - DEF(price, 8)
- Usage nodes:
  - USE(price, 6)
  - USE(price, 7)

Therefore, there are four du-paths:

- <5, 6>
- <5, 6, 7>
- <8, 9, 6>
- <8, 9, 6, 7>

All of these paths are definition-clear, so they are all dc-paths.

A naïve method of generating du-paths would be to use the Cartesian product of the set of defining nodes with the set of usage nodes. This is because this will often result in a number of infeasible paths: that is, paths that cannot be followed.

Rapps-Weyuker Metrics

Associated with the concepts discussed in the previous section are a set of test coverage metrics, also defined by Sandra Rapps and Elaine Weyuker in the early 1980s [2]. The metrics – a set of criteria, essentially – allow the tester to select sets of paths through the program, where "the number of paths selected is always finite, and chosen in a systematic and intelligent manner in order to help us uncover errors".

Paths through the program are selected, and test data – to be input into the program – is also selected to cover these paths (the percentage of coverage according to the set of paths selected). Having the set of paths contain all possible paths of the program (known as the All-Paths criterion, according to the Rapps/Weyuker nomenclature) is often infeasible, as the number of loops possible through the program – and therefore the number of potential paths to test – can often be infinite.

Nine criteria have been defined in the literature. Three correspond to the metrics used in path testing, where the paths selected are not chosen according to their variables and their attributes, but rather by an analysis of the structure of the program. These metrics are known as All-Paths (which has already been mentioned above), All-Edges and All-Nodes. All-Paths, which corresponds to the concept of 'path coverage', is satisfied if every path of the program graph is covered in the set. All-Edges, which corresponds to 'branch coverage', is satisfied if every edge (branch) of the program graph is covered. All-Nodes, which corresponds to 'statement coverage', is satisfied if every node is covered by the set of paths. In addition to these metrics, six new metrics were defined: All-DU-Paths, All-Uses, All-C-Uses/Some-P-Uses, All-P-Uses/Some-C-Uses, All-Defs and All-P-Uses. Definitions (adapted from the definitions in [2, 1]) of these metrics are provided below:

- The set of paths satisfies All-Defs for P if and only if, within the set of paths chosen, every defining node for each variable in the program has a definition-clear path to a usage node for the same variable, within the set of paths chosen.
- The set of paths satisfies All-P-Uses for P if and only if, within the set of paths chosen, every defining node for each variable in the program has a definition-clear path to every P-use node for the same variable.
- The set of paths satisfies All P-Uses/Some C-Uses for P if and only if, within the set of paths chosen, every defining node for each variable in the program has a definition-clear path to every P-use node for the same

variable: however, if there are no reachable P-uses, the definition-clear path leads to at least one C-use of the variable.

- The set of paths satisfies All C-Uses/Some P-Uses for P if and only if, within the set of paths chosen, every defining node for each variable in the program has a definition-clear path to every C-use node for the same variable: however, if there are no reachable C-uses, the definition-clear path leads to at least one P-use of the variable.

- The set of paths satisfies All-Uses for P if and only if, within the set of paths chosen, every defining node for each variable in the program has a definition-clear path to every usage node for the same variable.

- The set of paths satisfies All-DU-Paths for P if and only if, the set of paths chosen contains every feasible DU-path for the program.

Different criteria are supplied so that the tester can make what is described by Rapps and Weyuker as a "tradeoff" [2]. Although, in an ideal world, a program would be tested as thoroughly and 'completely' as possible – for example, with respect to structural testing, each and every possible combinations of nodes, branches, conditions, etc. would be tested thoroughly with every feasible combination of test data – in reality, a number of factor impede on this. For instance: time constraints; financial constraints; a situation where all 'major' areas of the system under test have been deemed to have been tested satisfactorily; or even the level of criticality – is the program's stability and reliability a critical factor (for instance, would lives be threatened if an error occurred in the program? Yes, if the program is controlling an aeroplane; no, if the program is controlling the in-flight games system for passengers!). Rapps and Weyuker have defined their "strongest" criterion to be All-DU-Paths; Jorgensen states that "the generally accepted minimum [is] All-Edges" [1].

Rapps and Weyuker noted that there was a relationship between the different metrics: certain metrics expanded upon other metrics – that is, if a set of paths satisfied a certain metric, then it also satisfied all the other metrics below it (for example, if All-Paths is satisfied, then so are All-DU-Paths and All-Uses). A diagram, created by Rapps and Weuyker, showing the relationship between metrics is shown in figure 4. This relationship was later described by Clarke *et al.* [3] as "subsumption".

In the diagram (figure 4), the arrows show the relationship between metrics. For example, All-Paths subsumes (or is *stronger than*) All-DU-Paths. However, Rapps and Weyuker describe that, during the development of the metrics, they had found that All-Defs is "not necessarily" stronger than All-Edges and

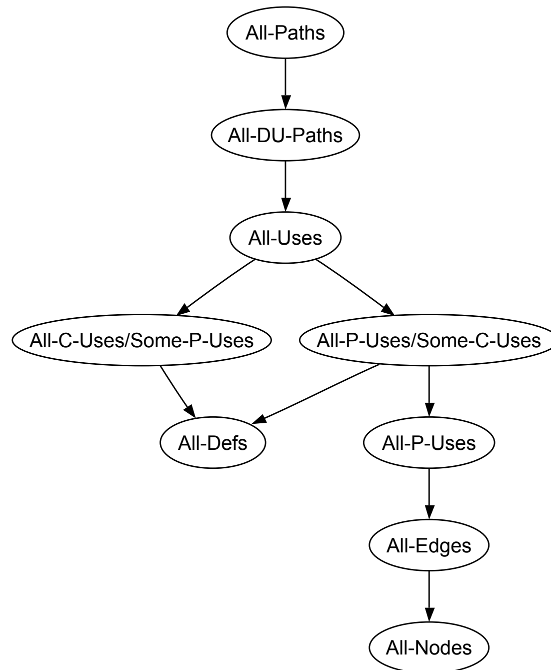

Figure 4: The Rapps-Weyuker Metrics [2]

All-Nodes: “in fact, the criteria are “incomparable” in the sense that it is possible for a given def-use graph G and sets of complete paths P1 and P2, that P1 satisfies all-edges, but not all-defs for G, while P2 satisfies all-defs, but not all-edges. Similarly, all-nodes and all-defs are shown to be incomparable.” [2].

This ‘incomparability’ meant that different criteria had to be found: at this point, Rapps and Weyuker established that there were two types of usage node – computational use (C-use) and predicate use (P-use). Therefore the three criteria All-P-Uses, All-P-Uses/Some-C-Uses, All-C-Uses/Some-P-Uses were defined. According to Rapps and Weyuker, All-P-Uses/Some-C-Uses provided “a way to satisfy our goal which includes All-Defs and All-Edges, but requires fewer test cases, in general, than . . . All-Uses”. All-C-Uses/Some-P-Uses was defined to allow tests to be defined where the “emphasis [is[on the flow of data rather than the flow of control (as in the case of all-p-uses/some-c-uses)” [2].

Tool support

This aspect of data flow testing is particularly suited to a level of automation using a software tool: indeed, Rapps and Weyuker state in their paper that “determining whether or not the criteria have been fulfilled can be checked

for mechanically. That is, we can write a program which can determine for a given program, test set, and selection criterion, whether or not the paths that would be traversed by the test set satisfy the criterion" [2]. However, outside the academic world, the proliferation of these tools does not seem to have taken off.

A search of academic journals has found two software tools that can automate (to a degree) the data flow testing process. The first, written by, amongst others, Elaine Weyuker and Phyllis Frankl, is called ASSET. ASSET works with programs written in the Pascal programming language [4]. Another data flow coverage tool, for the C programming language, was written by J. R. Horgan and S. London at Bellcore [5].

4 Program Slices

The concept of program slicing was first proposed by Mark Weiser in the early 1980s [6, 7]. According to Weiser, "slicing is a source code transformation of a program" [6], which allows a subset of a program, corresponding to a particular behaviour, to be looked at individually. This gives the benefit that a "programmer maintaining a large, unfamiliar program" does not have to understand "an entire system to change only a small piece" [6]. The concept of program slicing was extended to cover software maintenance by Keith Gallagher and James Lyle in 1991 [8], extending slices to become "independent of line numbers". Amended definitions of the program slice concept are given in Paul Jorgensen's book [1].

A program slice with respect to a variable at a certain point in the program, is the set of program statements from which the value of the variable at that point of the program is calculated. This definition can be amended to encompass the program graph concept: by replacing the set of program statements with nodes of the program graph. This allows the tester to find the list of usage nodes from the graph, and then generate slices with them.

Program slices use the notation S(V, n), where S indicates that it is a program slice, V is the set of variables of the slice and n refers to the statement number (i.e. the node number with respect to the program graph) of the slice.

So, for example, with respect to the `price` variable given in the example in section 2, the following are slices for each use of the variable:

- S(price, 5) = {5}
- S(price, 6) = {5, 6, 8, 9}
- S(price, 7) = {5, 6, 8, 9}
- S(price, 8) = {8}

To generate the slice S(price, 7), the following steps were taken:

- Lines 1 to 4 have no bearing on the value of the variable at line 7 (and, for that matter, for no other variable at any point), so they are not added to the slice.
- Line 5 contains a defining node of the variable `price` that can affect the value at line 7, so 5 is added to the slice.
- Line 6 can affect the value of the variable as it can affect the flow of control of the program. Therefore, 6 is added to the slice.

- Line 7 is *not* added to the slice, as it cannot affect the value of the variable at line 7 in any way.

- Line 8 is added to the slice – even though it comes after line 7 in the program listing. This is because of the loop: after the first iteration of the loop, line 8 will be executed before the next execution of line 7. The program graph in figure 1 shows this in a clear way.

- Line 9 signifies the end of the loop structure. This affects the flow of control (as shown in figure 1, the flow of control goes back to node 6). This indirectly affects the value of price at line 7, as the value stored in the variable will have almost certainly been changed at line 8. Therefore, 9 is added to the slice.

- No other line of the program can be executed *before* line 7, and so cannot affect the value of the variable at that point. Therefore, no other line is added to the slice.

The program slice, as already mentioned, allows the programmer to focus specifically on the code that is relevant to a particular variable at a certain point. However, the program slice concept also allows the programmer to generate a lattice of slices: that is, a graph showing the subset relationship between the different slices. For instance, looking at the previous example for the variable `price`, the slices S(price, 5) and S(price, 8) are subsets of S(price, 7).

With respect to a program as a whole, certain variables may be related to the values of other variables: for instance, a variable that contains a value that is to be returned at the end of the execution may use the values of other variables in the program. For instance, in the main example in this document, the `finalPrice` variable uses the `totalPrice` variable, which itself uses the `price` variable. The `finalPrice` variable also uses the `discount` variable, which uses the `staffDiscount` and `totalPrice` variables – and so on. Therefore, the slices of the totalPrice and discount variables are a subset of the slice of the finalPrice variable at lines 17 and 18, as they both contribute to the value. This subset relationship 'ripples down' to the other variables, according to the use-relationship described.

This is shown visually in the following example:

- S(staffDiscount, 3) = {3}

- S(totalPrice, 4) = {4}

- S(totalPrice, 7) = {4, 5, 6, 7, 8}

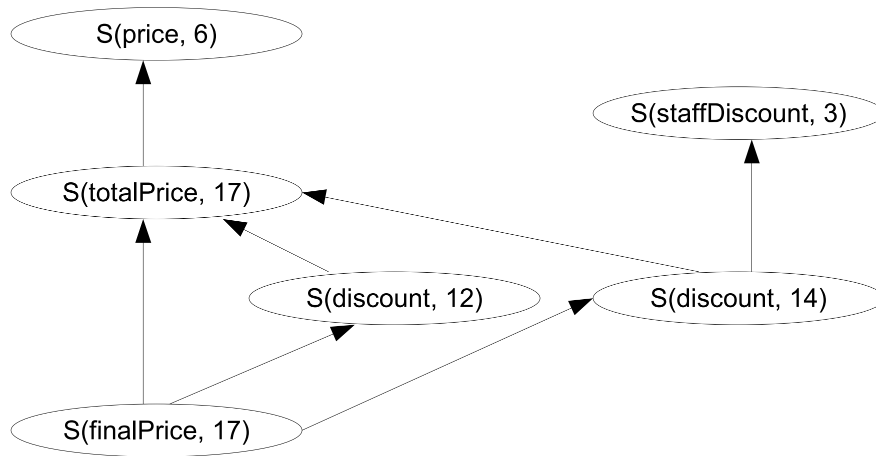

Figure 5: The Program Slice Lattice

- S(totalPrice, 11) = {4, 5, 6, 7, 8}
- S(discount, 12) = {3, 4, 5, 6, 7, 8, 11, 12}
- S(discount, 14) = {3, 4, 5, 6, 7, 8, 13, 14}
- S(finalPrice, 17) = {3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 17}

Therefore, the lattice of slices for the `finalPrice` variable is as shown in figure 5. This relationship, as shown in the lattice diagram, can feasibly help during testing, particularly if there's a fault. For instance, if there is an error in the slice of `finalPrice`, then, by testing the different subset slices, you can eliminate them from the possible sources of the error (for instance, the error may be generated from an incorrect calculation of the discount, for instance). If there is no error in the subset slices, then the error must be found in the remaining lines of code. As it is a set of statement fragments, this means that the remaining lines of code are the relative complement of the slice. In other words, the error is likely to be in:

Fullslice − SubsetSlices

If there *is* an error, then there could be errors in either the subsets, the code or both.

The relationship between slices also shows the interactions between variables in the code: if a slice for a variable `x` is a subset of a slice for a variable `y`, then the value of `x` must be needed by `y`. By generating the lattice, the tester can hopefully discover any unnecessary or undesired interactions between variables.

5 Conclusion

The concepts of define/use testing and slice a program 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 and use being correct (as well as the values assigned to the variable), data flow testing can be viewed as a useful addition to the tester's toolbox.

However, there are real constraints on the amount of testing that can be achieved. Much of the software in use everyday – be it in the most obvious location, on a computer, to the less obvious, such as a modern automatic washing machine with variable programme, spin cycle and temperature controls, to the control and auto-pilot systems in a modern commercial airliner (as used on trans-Atlantic flights, say) – is created, programmed and tested by commercial enterprises. These companies will often have very tight time and budget constraints, which will limit the amount of testing that can be achieved. Therefore, compromises have to be made. The temptation may be to focus more on the functional (black box) aspects of testing, without focussing as much on the structural (white box) testing. This may produce some results; however, it can be argued that, without a detailed knowledge of the structure of the software system, faults may be missed that can't be accounted for by merely testing the accessible interface of the system.

It is possible to focus on the physical structure (loops and branches, for instance) of the program, using path testing. However, there may still be faults that are missed with this, and the 'strongest' metric, path coverage, can often be impossible to achieve. The next strongest metric, branch coverage, is described as "the generally accepted minimum" metric by Jorgensen [1], when considered alongside data flow testing metrics and methods. Therefore, some degree (different metrics have been provided by Rapps and Weyuker to establish minimum achievable standards) of data flow testing should be considered as part of an overall testing strategy, along with program slicing techniques which, especially on larger projects with large teams of developers can help with understanding the workings of the code (for both the tester and developer), and hopefully increase the number of faults that are detected in the system during testing. Whilst adopting a data testing metric and program slicing techniques may not detect all faults, the fact that it can help to detect faults 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.

References

[1] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*. CRC Press, 2nd ed., 2002.

[2] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information.," *IEEE Trans. Software Eng.*, vol. 11, no. 4, pp. 367–375, 1985.

[3] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A formal evaluation of data flow path selection criteria.," *IEEE Trans. Software Eng.*, vol. 15, no. 11, pp. 1318–1332, 1989.

[4] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria.," *IEEE Trans. Software Eng.*, vol. 14, no. 10, pp. 1483–1498, 1988.

[5] J. R. Horgan and S. London, "Data flow coverage and the C language.," in *Symposium on Testing, Analysis, and Verification*, pp. 87–97, 1991.

[6] M. Weiser, "Program slicing.," in *ICSE*, pp. 439–449, 1981.

[7] M. Weiser, "Program slicing.," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, 1984.

[8] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance.," *IEEE Trans. Software Eng.*, vol. 17, no. 8, pp. 751–761, 1991.