**SE463 Assignment #3 Report**

Putting Together the Pieces of Data Flow Testing:

Taking a Look at Code Coverage Impacts

By: Elizabeth Dixon

**Table of Contents**

* Pg. 3: Project Overview
* Pg. 4-5: Organization of the Study

* Pg. 6-7: Findings Analysis & Discussion
* Pg. 8-9 : Discussion/Recap

**Project Overview**

Of the software testing methods we've studied and carried out throughout this class so far, it has been difficult to use any of them that are practical to do by hand. Boundary Value Analysis (BVA) had plentiful and redundant test cases prone to human error and Equivalence Class Testing (ECT) and Decision Table-based Testing (DTT) were a step up in practicality but were very challenging to partition and then organize consistently. However, while working on Data Flow Testing (DFT)/Path Testing, it is much easier to work practically in terms of the technical elements of the code provided. Maybe it's the difference between being able to see the code in Path-based testing techniques as opposed to only working with the executable in ECT and DTT, but as far as being able to understand the inner-workings (versus only outside-in design nuances) of the program, the patterns in behavior that could help constitute test cases are much clearer. Going through the motions of this project, overall, gave me a better understanding of how important it is to write consistent code so that pattern recognition, and therefore testing, can be feasible. I cannot imagine going through the program provided if the variables were always declared in abnormal places or the functions were called in different orders each time and so on.

**Organization of the Study**

The first thing I had to do to begin my Path Testing-based understanding of this program was to create a program graph. For simplicity's and organization's sake, I went in sequential order of the program. However, not necessarily in the way that the functions are called in the main function, as those function calls could be referenced down the line when they are written out in the graph (which will make sense as you view "SE463\_A3\_Program\_Graph.pdf"). I thought that it would be too jumbled and that the reasoning and order of the main function would be lost if I had done so. What I mean by this is, I started by looking at how the patterns of conditionals and loops look in node/program graph form and wrote out nodes that would both help guide the viewer of my program graph as well as act as the practical nodes for the supplemental definitions (such as DEF, USE, P-Use, and C-use nodes and their respective DU-/DC-paths). Writing out the program graph was the most time-consuming part of this project for me, as I had to reference both the program itself to make sure the logic and coordination of line numbers was correct for each node and the textbook's conditional pattern examples to make sure that my program graph would be readable for the viewer. After drafting up the program graphs for each function (the main, advancePlayer, and printPosition functions), I decided that it would be better as a deliverable to transfer the work into Lucidchart. This way, the flow of the program and the thought process behind my organization would be clearer (as opposed to providing screenshots of the graphs loosely sketched on paper) and so that I could label each node and path as necessary. With constructing the program graph out of the way, it can now be used as a solid reference for identifying the nodes' types and defining them.

**Findings Analysis & Discussion**

By looking at the program as well as the program graph, Jonathan was able to begin the definition process in terms of the DEF, USE, P-Use, and C-Use nodes. When reviewing his work, I kept in mind all the possibilities of what DEF and USE nodes could be - like remembering that the variables in an output statement and that variables that are used to assign a value to another variable are USE nodes, to name a couple. Additionally, the program graph was even more useful for defining the P-Use and C-Use nodes - keeping in mind that they are defined by their outdegrees. Separately, I was responsible for coming up with examples of possible DU-Paths and DC-Paths for every variable in the program. The applicable list of variables came down to playerA (conditionally referenced by ptrplayerA), playerB (conditionally referenced by ptrplayerB), obstacalProb, and GAME\_LENGTH. The examples of paths that I was able to come up with were fairly straightforward with an exception or two. One thing I had to realize was that this time, I should truly focus on how the program would actually run (i.e. when looking at a function call in the main function, remembering to go back to the main function to look at the next lines of code after looking at a function *definition* later in the program). The only difficult path to define was obstacalProb's DC-Path. Unlike the other variables (GAME\_LENGTH aside), it was not defined in the main function, and was defined in advancePlayerA and advancePlayerB's scopes as 0. Then, the line that was always after its definition was another defining node of obstacalProb, so there could not be a fleshed out DC-Path for it because another node in its path besides its initial defining node would be a defining node.

**Discussion/Recap**

So, not only does the DFT software testing technique provide high testing coverage for the program itself, it can tell the tester how effective each variable's path is at covering the program. This aspect of note could also pertain to creating and analyzing test cases out of the organization described in previous sections. Not only would variable code coverage be a useful metric to determine how effective this method of software testing would be for this scenario (or parts of the scenario), it would also be a good deliverable to a team out in industry. For example, playerA's DU-Path (considering each node as its respective line of code for simplicity's sake) would cover around 17/173 or 9.8% of the program if 30 lines of code are taken out from the program's total of 203 to account for the #include s, their spaces, and labeling comments. If this calculation is done for each variable and each possible path, software testers could tell developers how to optimize their code, for example, and give them an idea of how much coverage a variable's path would have for other similar conditionals so that predictions can be made about what coding/organizational methods would be best to use.

Just from analyzing this short "Candy\_Solution.cpp" program alone, as mentioned before, I can already discern so many possibilities of it being used in practical ways - out in industry and otherwise (even if done by hand). Not only are the steps to get to a final product while setting up the testing method great for visualization (while BVA, ECT, and DTT can struggle to do that - even with a spreadsheet), there are multiple ways to define each main element of the program (i.e. node typing). This could be advantageous when working on a team - to have multiple ways to describe one entity, that is. Even in a non-technical way, a program graph when made with the appropriate focus can be readable for any user or viewer of a project in question's work, which I feel is very important for accessibility purposes.