Cyclomatic Complexity UCC-J

University of Southern California

Center for Systems and Software Engineering

December , 2016

Revision Sheet

Date	Version	Revision Description	Author
2/18/2016	1.0	Original Release	Matthew Swartz

Table of Contents

1. Introduction

Thomas McCabe developed a measure of program complexity in 1976, referred to as cyclomatic complexity. Cyclomatic complexity counts the number of linearly independent paths within a program. Effort required in code construction is affected by complexity which is why it is a valuable metric incorporated in the Unified Code Counter Government edition (UCC-G) tool. Additionally, developers are able to determine the number of independent path executions and baseline unit tests required for validation. Being aware of the cyclomatic complexity, developers find that they can assure that all paths have been tested at least once. The details of how the McCabe cyclomatic complexity metric is measured in UCC-G will be described with examples in later sections of this document. This standard applies across all of the languages for which cyclomatic complexity has been implemented.

2. McCabe Cyclomatic Complexity

As mentioned previously, cyclomatic complexity is a count of the number of linearly independent paths within a program. For instance, a simple linear program that has no decision points has a complexity of 1, whereas if it contained an IF statement, then there are two separate paths through the code and so it would have a complexity of 2.
An easy way to calculate cyclomatic complexity is to take the number of decisions being made in the code and adding 1. The specific word and syntax depends on the programming language, but the following keywords usually identify decision points within code:

- IF
- ELSE IF
- REPEAT-UNTIL
- WHILE
- FOR
- CASE
- Database exception clause (except for when successful)

3. Cyclomatic Complexity Rings and Implementation

USC's Information Engineering Technology program did a study of the cyclomatic complexity for a specific domain of projects, and has described cyclomatic complexity in 4 "rings" CC1 – CC4. This categorization of cyclomatic complexity has allowed for the implementation to be broken down as follows:

a. CC1: The original McCabe method treated each branch as a count.
b. CC2: A variation of the original method that counts Boolean operators within the decision point
   1. For example, the statement

      IF a==1 AND b==1

      Would receive a cyclomatic complexity for each value 1 for CC1, but received a calue 2 in compliance with CC2
c. CC3: Increments cyclomatic complexity for each CASE OF clause, and ignores the individual CASE clauses.
d. CC4: Counts distinct IF/ELSE IF statements. Hence, if an IF statement is present multiple times within a function or file, it is only counted once

The following table summarizes the differences between the four cyclomatic complexity rings:

STATEMENT TYPE	CC1	CC2	CC3	CC4
IF/ELSE IF	+1	+1 for IF/ELSE IF and +1 for each AND/OR clause	+1	+1 for each distinct IF/ELSE IF clause
REPEAT-UNTIL	+1	+1	+1	+1
WHILE	+1	+1	+1	+1
FOR	+1	+1	+1	+1
CASE OF	0	0	+1	0
CASE	+1	+1	0	+1
Database exception clause	+1	+1	+1	+1

Table 1 – Differences between Cyclomatic Complexity Rings

The cyclomatic complexity measurement for CC1, CC2 and CC3 are reported as separate columns in the UCC Cyclomatic Complexity Output report "outfile_cyclomatic_cplx.csv". The results are reported by file, as well as function/module. CC1 is the only ring of complexity that has been implemented within UCC-J.

FILE LEVEL CYCLOMATIC COMPLEXITY

In scripting languages such as Python or Matlab, it is possible to have cyclomatic complexity outside of a function or class. An example of this can be found in the following Matlab script

```
for m = 1:5
    for n = 1:100
        b = MyFunction(m * n)
    end
end

function y = MyFunction(x)
   while (x > 10)
      y = x / 2;
   end
end
```

The function `MyFunction` has a cyclomatic complexity of 2, but this script has cyclomatic complexity at the file level of 4.

NESTED FUNCTION CYCLOMATIC COMPLEXITY

The current implementation of how cyclomatic complexity is counted in the case where there are nested functions is that each function's cyclomatic complexity is counted independent of the others (i.e. the counts do not roll up into the outermost function. An example of this can be found in the following Matlab script

```
function main
   nestedfun1
   nestedfun2

   for a = [24,18,17,23,28]
```

```
      disp(a)
   end

   function nestedfun1
      x = 1;
      for a = [24,18,17,23,28]
         disp(a)
      end

         function nestedfun3
            x = 2;
            for a = [24,18,17,23,28]
               disp(a)
            end
         end
   end

   function nestedfun2
      x = 2;
      for a = [24,18,17,23,28]
         disp(a)
      end
   end
end
```

In this script we have 4 functions (in nested order):

- `main`
  - `nestedfun1`
    - `nestedfun3`
  - `nestedfun2`

The current implementation will give cyclomatic complexity counts as

- `nestedfun3` – 2
- `nestedfun1` – 2
- `nestedfun2` – 2
- `main` – 2

with a total file level cyclomatic complexity of 8.

4. Risk Evaluation

SEI has identified a risk evaluation or level with cyclomatic complexity ranges – this risk evaluation is provided in the UCC-G Cylcomatic Complexity Output report "outfile_cyclomatic_cplx.csv":

Cyclomatic Complexity	Risk Evaluation
1 – 10	A simple program, without much risk
11 – 20	More complex, moderate risk
21 – 50	Complex, high risk program
> 50	Untestable program (very high risk)

Table 2 – Risk Evaluation of Cyclomatic Complexity Levels