Essential complexity

Essential complexity is a numerical measure defined by Thomas J. McCabe, Sr., in his highly cited, 1976 paper better known for introducing cyclomatic complexity. McCabe, defined essential complexity as the cyclomatic complexity of the reduced CFG (control flow graph) after iteratively replacing (reducing) all structured programmingcontrol structures, i.e. those having a single entry point and a single exit point (for example if-then-else and while loops) with placeholder single statements.[1]:317[2]:80

McCabe's reduction process is intended to simulate the conceptual replacement of control structures (and actual statements they contain) with subroutine calls, hence the requirement for the control structures to have a single entry and a single exit point.[1]:317 (Nowadays a process like this would fall under the umbrella term of refactoring.) All structured programs evidently have an essential complexity of 1 as defined by McCabe because they can all be iteratively reduced to a single call to a top-level subroutine.[1]:318 As McCabe explains in his paper, his essential complexity metric was designed to provide a measure of how far off this ideal (of being completely structured) a given program was.[1]:317 Thus greater than 1 essential complexity numbers, which can only be obtained for non-structured programs, indicate that they are further away from the structured programming ideal.[1]:317

To avoid confusion between various notions of reducibility to structured programs, it's important to note that McCabe's paper briefly discusses and then operates in the context of a 1973 paper by S. Rao Kosaraju, which gave a refinement (or alternative view) of the structured program theorem. The seminal 1966 paper of Böhm and Jacopini showed that all programs can be [re]written using only structured programming constructs, (aka the D structures: sequence, if-then-else, and while-loop), however, in transforming a random program into a structured program additional variables may need to be introduced (and used in the tests) and some code may be duplicated.[3]

In their paper, Böhm and Jacopini conjectured, but did not prove that it was necessary to introduce such additional variables for certain kinds of non-structured programs in order to transform them into structured programs.[4]:236 An example of program (that we now know) does require such additional variables is a loop with two conditional exits inside it. In order to address the conjecture of Böhm and Jacopini, Kosaraju defined a more restrictive notion of program reduction than the Turing equivalence used by Böhm and Jacopini. Essentially, Kosaraju's notion of reduction imposes, besides the obvious requirement that the two programs must compute the same value (or not finish) given the same inputs, that the two programs must use the same primitive actions and predicates, the latter understood as expressions used in the conditionals. Because of these restrictions, Kosaraju's reduction does not allow the introduction of additional variables; assigning to these variables would create new primitive actions and testing their values would change the predicates used in the conditionals. Using this more restrictive notion of reduction, Kosaraju proved Böhm and Jacopini's conjecture, namely that a loop with two exits cannot be transformed into a structured program *without introducing additional variables*, but went further and proved that programs containing multi-level breaks (from loops) form a hierarchy, such that one can always find a program with multi-level breaks of depth *n* that cannot be reduced to a program of multi-level breaks with depth less than *n*, again without introducing additional variables.[4][5]

McCabe notes in his paper that in view of Kosaraju's results, he intended to find a way to capture the essential properties of non-structured programs in terms of their control flow graphs.[1]:315 He proceeds by first identifying the control flow graphs corresponding to the smallest non-structured programs (these include branching into a loop, branching out of a loop, and their if-then-else counterparts) which he uses to formulate a theorem analogous to Kuratowski's theorem, and thereafter he introduces his notion of essential complexity in order to give a scale answer ("measure of the structuredness of a program" in his words) rather than a yes/no answer to the question of whether a program's control flow graph is structured or not.[1]:315 Finally, the notion of reduction used by McCabe to shrink the CFG is not the same as Kosaraju's notion of reducing flowcharts. The reduction defined on the CFG does not know or care about the program's inputs, it is simply a graph transformation.[6]

For example, the following C program fragment has an essential complexity of 1, because the inner **if** statement and the **for** can be reduced, i.e. it is a structured program.

```
   for (i = 0; i < 3; i++) {
      if (a[i] == 0) b[i] += 2;
   }
```

The following C program fragment has an essential complexity of four; its CFG is irreducible. The program finds the first row of z which is all zero and puts that index in i; if there is none, it puts -1 in i.

```
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++) {
         if (z[i][j] != 0)
            goto non_zero;
      }
      goto found;
non_zero:
   }
   i = -1;
found:
```

The idea of CFG reducibility by successive collapses of sub-graphs (ultimately to a single node for well-behaved CFGs) is also used in modern compiler optimization. However the notion from structured programming of single-entry and single-exit control structure is replaced with that of natural loop, which is defined as a "single-entry, multiple-exit loop, with only a single branch back to the entry from within it". The areas of the CFG that cannot be reduced to natural loops are called *improper regions*; these regions end up having a fairly simple definition: multiple-entry, strongly connected components of the CFG. The simplest improper region is thus a loop with two entry points. Multiple exits do not cause analysis problems in modern compilers. Improper regions (multiple-entries into loops) do cause additional difficulties in optimizing code.[7]

See also

- History of software engineering
- Decision-to-decision path
- Cyclomatic complexity

References

1. McCabe (December 1976). "A Complexity Measure"*IEEE Transactions on Software Engineering* 308–320. doi:10.1109/tse.1976.233837(https://doi.org/10.1109%2Ftse.1976.233837)
2. http://www.mccabe.com/pdf/mccabe-nist235r.pdf
3. David Anthony Watt; William Findlay (2004).*Programming language design concepts* John Wiley & Sons. p. 228. ISBN 978-0-470-85320-7
4. S. Rao Kosaraju (December 1974). "Analysis of structured programs"*Journal of Computer and System Sciences* **9** (3): 232–255. doi:10.1016/S0022-0000(74)80043-7(https://doi.org/10.1016%2FS0022-0000%2874%2980043-7)
5. For more modern treatment of the same results see: Kozen,The Böhm–Jacopini Theorem is False, Propositionally (http://www.cs.cornell.edu/~kozen/papers/bohm-jacopini.pdf)
6. McCabe footnotes the two definitions of on pages 315 and 317.
7. Steven S. Muchnick (1997).*Advanced Compiler Design Implementation* Morgan Kaufmann. pp. 196–197 and 215. ISBN 978-1-55860-320-2