A Simple and Effective Measure for Complex Low-Level Dependencies

Dirk Beyer
Simon Fraser University, B.C., Canada
& University of Passau, Germany

Ashgan Fararooy
Simon Fraser University, B.C., Canada

Abstract

The measure dep-degree is a simple indicator for structural problems and complex dependencies on code-level. We model low-level dependencies between program operations as use-def graph, which is generated from reaching definitions of variables. The more dependencies a program operation has, the more different program states have to be considered and the more difficult it is to understand the operation. Dep-degree is simple to compute and interpret, flexible and scalable in its application, and independently complementing other indicators. Preliminary experiments suggest that the measure dep-degree, which simply counts the number of dependency edges in the use-def graph, is a good indicator for readability and understandablity.

1. Overview

Software systems, due to the frequency and amount of changes to their structure, are very different from artifacts in other engineering disciplines. The frequent changes —often essentially affecting the stability of the system— require a continuous effort to prevent the structure from degeneration. There are several theories why this must happen (e.g., [12]) and guidelines on how to prevent or fix this (e.g., Design Patterns, Refactoring, Beautiful Code).

We present a simple but effective idea that contributes in solving the following subproblem: Given two versions of a software program that have the same behavior and differ only in code structure, which version is to prefer in terms of low-level dependency structure. For example, if the second version is the result of changing the first version, we would like to know whether the change actually improves the code structure, i.e., was a positive 'refactoring'.

The indicator dep-degree (DD), which we define in the next section, is based on the notion of reaching definitions, a well-known concept from compiler optimization and program analysis. Dep-degree is defined for single program operations as well as for program functions. The DD for a single operation is the total number of reaching definitions for the variables that it uses. The DD for a set of operations (or a complete function) is the sum over all dep-degrees for operations (or the number of edges in the use-def graph, respectively). For example, consider the assignment operation $\texttt{x} = \texttt{a} - \texttt{b}$. The dep-degree for this operation is the number of different reaching definitions for variable `a` plus the number of different reaching definitions for variable `b`.

In Sect. 3, we compare dep-degree with the two most widely used —but not necessarily academically accepted— measures for software programs. The first measure is lines of code (LOC), an indicator for the size of a program. It measures the length of a program by a pure syntactical count of lines, without analyzing the contents in detail. [1] The second measure that we compare with is cyclomatic complexity (CC) [14], an indicator for the complexity of the control-flow structure of a program. It measures the control-flow structure by counting nodes, edges, and connected components ($CC(G) = e - n + p$, where e, n, and p are the number of edges, nodes, and connected components in control-flow graph G, respectively). Dep-degree (DD) is an indicator for complex low-level dependencies of a program. It measures the amount of data-flow dependencies by counting edges in the use-def graph.

Besides LOC and cyclomatic complexity, there is a rich set of software measures defined in the literature: there are classic software measures [2, 3, 5, 6, 7, 8, 9, 10, 16, 21, 22], most of them still in use today; there have been recent efforts to create new software measures that are supported by richer theoretical background (e.g., [11]); there are measures for object-oriented programming [13, 18]; and some that proposed to measure program complexity based on data-flow information (e.g., [21]). The indicators measure certain properties of software, attempting to indicate size, product properties, quality, and complexity. For our comparison, we chose LOC as the most prominent indicator for size, and cyclomatic complexity as the most prominent indicator for control-flow complexity. For details about the various measures we refer the reader to the survey and discussion articles on software measures [2, 3, 6, 7, 9, 10, 16].

[1] We distinguish between measure and indicator: a *measure* needs to be precise, and we use it in the spirit of Stevens [20], while we allow the weaker term *indicator* to be less precise.

978-0-7695-4113-6/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPC.2010.49

IEEE computer society

The application of software measures has significantly advanced the techniques to automatically and abstractly assess properties of large software systems. But many software engineers were too enthusiastic in applying measures, trying to use measures as indicators for properties that they were not designed for. For example, LOC was often considered a measure for size, but it is a measure for length, and just an indicator for size. Or, cyclomatic complexity was sometimes used as measure for program complexity, but it is a measure for cyclomatic complexity of control-flow graphs, and might only roughly indicate program complexity. This was extensively discussed in the literature, and there was criticism on measurement practice in the literature [9], and effort was spent on improving existing measures [17, 19], and then on showing the limitations of the extensions again [4].

2. Dependency Degree

Control-Flow Graph (CFG). We represent a computer program as a collection of control-flow graphs [1], one for each function (or procedure) of the program. A *control-flow graph* $G = (B, F)$ is a directed graph that consists of a set B of program operations (the nodes of the graph) and a set $F \subseteq B \times B$ of control-flow edges of the program. A program operation is executed when control moves from the entry to an exit of the operation node. A program operation is either an assignment operation, a conditional, a function call, or a function return. A conditional is a predicate that must be evaluated to true (false) for control to proceed along the first (second, resp.) exit edge. All other operations have one exit edge. Program operations can read and write values via variables from the set X of program variables.

Reaching Definitions. In this paper, we use a notion of operation dependency that is motivated by the data-flow analysis for reaching definitions [1]. The function *reaching definitions* $rd_G : B \times X \to 2^B$ for a CFG $G = (B, F)$ assigns to a program operation b_u and a variable x the set of all definitions of variable x that can reach the operation b_u. In other words, a program operation b_d is in the set of reaching definitions for program operation b_u and variable x, if b_d is an assignment operation or a function call that assigns a value to x and there exists a path in the CFG from b_d to b_u on which no other program operation assigns a value to x.

Use-Def Graph. We now derive the use-def graph from the results of the reaching-definitions analysis. A *use-def graph* $S_G = (B, E)$ for a CFG $G = (B, F)$ is a directed graph that consists of the set B of program operations (of G) and the set E of use-def edges that are derived from the reaching-definitions function as follows: an edge (b_u, b_d) is member of the set E if there exists a variable x that is used in b_u and for which $b_d \in rd_G(b_u, x)$ holds.

The use-def graph is a *dependency graph* on operation level, more precisely, it models the data-flow dependencies

```
void                          void
swap(int a, int b) {          swap(int a, int b) {
   a += b;                       int temp = a;
   b = a - b;                    a = b;
   a -= b;                       b = temp;
}                             }
```

Figure 1. Two 'swap' implementations

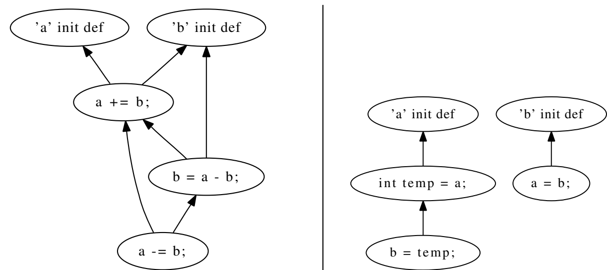

Figure 2. Use-def graphs for 'swap'

between operations and the direction of an edge models the direction of the dependency (from use to definition). In compiler optimization and program analysis, this data-flow dependency is one of the most important dependencies that are considered (but mostly stored in a different form as so-called ud-chains [1]).

Dependency-Degree. The *dep-degree for program operations* in a CFG $G = (B, F)$ is a total function $dd_G : B \to \mathbb{N}$ that assigns to each program operation b the number of other program operations that it depends on in $S_G = (B, E)$, i.e., $dd_G(b) = |\{b' \in B \mid (b, b') \in E\}|$ (the out-degree of b in graph S_G). The *dep-degree for program functions* is a total function $dd : \mathbb{G} \to \mathbb{N}$ that assigns to each control-flow graph G the number of edges in its dependency graph $S_G = (G, E)$, i.e., $dd(G) = \sum_{b \in B} dd_G(b) = |E|$ (the sum of all out-degrees in graph S_G).

Inspired by Miller's article on our capacity for processing information [15], we believe that the comprehension of program code is easy if we have to remember only a few possible states of the program (e.g., different variable values, branching choices), and that we make mistakes while programming, or misunderstand a program, if we have to remember too much information about the current program state. The dep-degree for a single program operation tells us how many different pieces of information we need to consider in order to understand the effect of the program operation; more precisely, it tells us the number of all different predecessor operations that influence the effect of the considered program operation (it sums up, over all variables used in the operation, the number of different reaching definitions). Thus, if Miller's insight is true for program understanding, then the dep-degree of an operation is a good indicator for the difficulty to understand the operation.

```
// Require: n >= k >= 0
int bico(int n, int k) {
  int[] arr = arrInit(n+1);
  for (int i = 0; i <= n; i++) {
    int temp = arr[0];
    for (int j = 1; j < i; j++) {
      arr[j] = arr[j] + temp;
      temp = arr[j] - temp;
    }
  }
  return arr[k];
}
```

```
// Require: n >= k >= 0
int bico(int n, int k) {
  int facK = 1;
  for (int i = 1; i <= k; i++) {
    facK = facK * i;
  }
  int facNk = 1;
  for (int j = n; j > n-k; j--) {
    facNk = facNk * j;
  }
  return facNk / facK;
}
```

Figure 3. Two 'bico' implementations

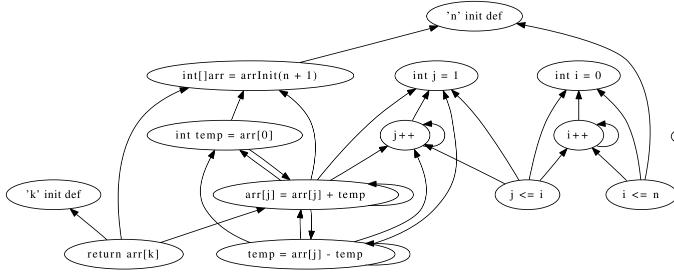

Figure 4. Use-def graph for 'bico' (left)

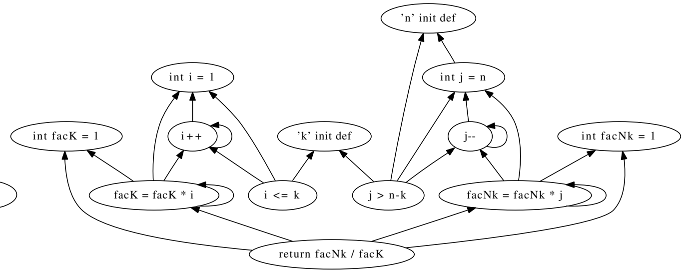

Figure 5. Use-def graph for 'bico' (right)

3. Examples

Assignments and Arithmetics. Consider the two implementations of the function `swap` in Fig. 1. The first implementation (left) has the advantage of using only two registers —which are allocated already anyway— but the disadvantage of being more difficult to understand because it uses not only assignments but also arithmetics. [2] The second implementation (right) has the advantage of being easy to understand —it uses only assignment operations— but the disadvantage that a simple code generator would allocate three registers for the execution of this code. Figure 2 shows the use-def graphs for the two swap functions (a node labeled 'init def' refers to the parameter initialization of the call-by-value). The graph layout was calculated using GRAPHVIZ (dot). On the right, the value of variable `b` (third assignment) depends on the assignment of variable `temp` which in turn depends on the initial value of variable `a`. The value of `a` depends on the initial value of `b`. The graph on the left illustrates that this implementation not only involves arithmetics, but also has a more complicated dependency structure. The DD is 6 for the function on the left and 3 for the function on the right, which indicates that the left function has a more complex dependency structure. Table 1 shows that LOC and CC do not distinguish the two functions, because LOC measures length and CC measures control-flow complexity, which is the same for both functions.

Strength Reduction and Nested Loops. In Fig. 3 we compare two implementations for computing binomial coefficients. Both functions take as input two non-negative integers `n` and `k` (required: $k \leq n$), and compute the binomial coefficient $\binom{n}{k}$ (n choose k). The function on the left computes the result without using multiplication — it simulates Pascal's triangle to perform the computation. The array `arr` contains the i-th row of the triangle at the end of the i-th iteration of the outer 'for' loop. The disadvantage of this program is that it is rather difficult to understand because it uses nested loops instead of a sequence of two loops, and it uses an array, the content of which is important to understand. (An array access is more difficult than a variable access because it involves the array pointer and an index.) The function on the right computes (almost directly) the result using the formula $\binom{n}{k} = \frac{n!}{(n-k)!k!}$, but has the disadvantage of using multiplication (more expensive to compute, more expensive to verify because not linear). We say 'almost directly' because the second 'for' loop calculates $n(n-1)\ldots(n-k+1) = \frac{n!}{(n-k)!}$. The two functions `bico` are equal in the number of lines of code (LOC), the number of statements, and the number of local variables (`i,j,temp,arr` versus `i,j,facK,facNk`). Furthermore, the functions use the same number of control structures (two 'for' loops), and therefore the cyclomatic complexity yields the same value for both functions. But the low-level dependency structures of the two functions are very different. The dependency graphs are shown in Figs. 4 and 5. The graph in Fig. 4 has higher density such that the graph-drawing algorithm 'dot' from GRAPHVIZ was not able to find a layout without edge crossings. In Table 1, LOC and cyclomatic complexity yield the same values, and dep-degree yields the values 28 and 24, respectively, for the two implementations, indicating that the first implementation has a more complex dependency structure.

[2] Furthermore, one has to understand the arithmetic-overflow semantics of the programming language in order to establish correctness.

```java
class Pair {
  int x;
  int y;
  boolean equals(Object o) {
    boolean result = false;
    if (o != null) {
      if (o instanceof Pair) {
        result = this == o;
        Pair p = (Pair) o;
        result = result ||
                 ( (x == p.x) && (y == p.y) );
      }
    }
    return result;
  }
}
```

```java
class Pair {
  int x;
  int y;
  boolean equals(Object o) {
    if (o == null) {
      return false;
    }
    if (this == o) {
      return true;
    }
    if (! (o instanceof Pair) ) {
      return false;
    }
    Pair p = (Pair) o;
    return (x == p.x) && (y == p.y);
  }
}
```

Figure 6. Two 'equals' implementations

Method	LOC	CC	DD
swap (left)	3	1	6
swap (right)	3	1	3
bico (left)	9	3	28
bico (right)	9	3	24
equals (left)	10	3	11
equals (right)	11	4	8

Table 1. Indicator values LOC, CC, and DD

Early Return. In the last example we consider two alternative implementations of the `equals` function for a class `Pair` (of two integer values). Figure 6 shows the example functions. The two functions follow the same logic, but the first implementation uses a local variable `result` to store the decision to return, whereas the second implementation returns as early as possible. The second implementation seems to be easier to understand, because all special cases are checked and immediately dealt with; after this, the reader can forget them, i.e., there are not many dependencies. The first implementation requires the reader to track the outcome of the various comparisons, and the last value of variable `result`, all the way to the end of the function. The cyclomatic complexity of the second implementation is higher, because it uses one more 'if' statement (cf. Table 1). The program length LOC prefers the first implementation, because it is shorter. The value of dep-degree witnesses that the dependency structure of the second implementation is less complicated (DD=8) than that of the first one (DD=11).

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] B. Curtis, S. B. Sheppard, and P. Milliman. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In *Proc. ICSE*, 1979.

[3] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Trans. Softw. Eng.*, 5(2):96–104, 1979.

[4] G. K. Gill and C. F. Kemerer. *Cyclomatic-complexity metrics revisited: An empirical study of software development and maintenance*. MIT, 1991.

[5] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier, 1977.

[6] S. M. Henry and D. G. Kafura. Software-structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 7(5):510–518, 1981.

[7] S. M. Henry, D. G. Kafura, and K. Harris. On the relationships among three software metrics. In *Proc. Measurement and Evaluation of Softw. Quality*, pages 81–88. ACM, 1981.

[8] S. S. Iyengar, N. Parameswaran, and J. Fuller. A measure of logical complexity of programs. *Computer Languages*, 7(3-4):147–160, 1982.

[9] C. Jones. Software metrics: Good, bad, and missing. *Computer*, 27(9):98–100, 1994.

[10] D. Kafura and G. R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Trans. Softw. Eng.*, 13(3):335–343, 1987.

[11] S. R. Kirk and S. Jenkins. Information theory-based software metrics and obfuscation. *J. Systems and Software*, 72(2):179–186, 2004.

[12] M. M. Lehman and L. A. Belady. *Program evolution: Processes of software change*. Academic Professional, 1985.

[13] W. Li. Another metric suite for object-oriented programming. *J. Systems and Software*, 44(2):155–162, 1998.

[14] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.

[15] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.

[16] E. E. Mills. *Software Metrics*. Curriculum Module SEI-CM-12-1.1, CMU-SEI, 1988.

[17] G. J. Myers. An extension to the cyclomatic measure of program complexity. *SIGPLAN Notices*, 12(10):61–64, 1977.

[18] S. Purao and V. K. Vaishnavi. Product metrics for object-oriented systems. *ACM Comp. Surv.*, 35(2):191–221, 2003.

[19] F. Stetter. A measure of program complexity. *Computer Languages*, 9(3-4):203–208, 1984.

[20] S. S. Stevens. On the theory of scales of measurement. *Science*, 103(2684):677–680, 1946.

[21] K.-C. Tai. A program-complexity metric based on data-flow information in control graphs. In *Proc. ICSE*, pages 239–248. IEEE, 1984.

[22] M. R. Woodward, M. A. Hennell, and D. Hedley. A measure of control-flow complexity in program text. *IEEE Trans. Softw. Eng.*, 5(1):45–50, 1979.