A Formal Evaluation of DepDegree Based on Weyuker's Properties

Dirk Beyer
University of Passau, Germany

Peter Häring
University of Passau, Germany

ABSTRACT

Complexity of source code is an important characteristic that software engineers aim to quantify using static software measurement. Several measures used in practice as indicators for software complexity have theoretical flaws. In order to assess the quality of a software measure, Weyuker established a set of properties that an indicator for program-code complexity should satisfy. It is known that several well-established complexity indicators do not fulfill Weyuker's properties. We show that DepDegree, a measure for data-flow dependencies, satisfies all of Weyuker's properties.

Categories and Subject Descriptors: D.2.8 [*Software Engineering*]: Metrics — Complexity Measures
General Terms: Measurement, Theory
Keywords: Software Measurement, Complexity Measure

1. INTRODUCTION

Software measurement is a static analysis technique that software engineers and maintainers apply to obtain a quick overview over certain characteristics of the software that they inspect. There are many software measures available, for different levels of granularity, and for different aspects of the software. One of the (open) problems in software measurement is to correctly quantify the complexity of source code. It is known that difficult-to-understand program code hinders efficient software maintenance, and that refactoring [5] can help to transform difficult program code into program code that is easier to understand and maintain.

A complexity measure m for program code is a function that assigns to each program (in the real world) a numeric value (in the formal world) and that fulfills the following property (among others): Given two programs p_1 and p_2, if p_1 is more complex than p_2, then the value $m(p_1)$ is larger than the value $m(p_2)$. Such a measure does not exist in the literature. This is mainly due to the fact that complexity of program code is not well defined. Several *indicators* for program complexity and understandability exist, i.e., functions that (exactly) measure certain aspects of the software. These are used in practice to get a rough idea about program-code complexity. Empirical studies are required to assess the quality of an indicator for program complexity.

There are several examples for software measures that are used in practice as indicators for software complexity: the *cyclomatic complexity* [9] measures the number of linearly independent paths in the control-flow graph of a program, the *statement count* measures the length of a program in terms of program statements, the *Halstead effort* [7] measures an arithmetic combination of the vocabulary of operators and operands that are used in the program, and *DepDegree* [1] measures the number of data-flow dependencies in a program.

As part of a larger body of work to evaluate software measures, in this paper we are interested in a systematic investigation of the *formal quality* of an indicator that was not yet formally investigated in the literature before: *DepDegree* [1]. Formal evaluations of measures should be performed in addition to empirical evaluations, in order to ensure theoretical soundness of the measurement results and their interpretation. We are using Weyuker's nine properties that an indicator for program complexity should satisfy [16]. Weyuker's properties encode a large amount of community knowledge and experience on what experts consider desired properties of measures to indicate program complexity. It is known that all other above mentioned and widely-used indicators do not fulfill some of the nine properties [13, 16]. The contribution of this paper is to show that DepDegree fulfills all nine properties. Weyuker's properties have also been applied and discussed in the context of object-oriented measures (e.g., [3, 6]), component-based development [15], and business-process complexity measures [2].

2. BACKGROUND

Weyuker's Properties. Elaine J. Weyuker proposed a set of nine properties that indicators for software complexity should possess [16]. She evaluated, using those properties, the measures statement count, Halstead effort [7], cyclomatic complexity [9], and Oviedo's data-flow complexity [14]. Her analysis showed that none of the evaluated measures fulfills all nine properties. The properties are *not* meant as a conclusive evaluation of software measures, but as *formal prerequisites* of measures to satisfy an intuitive understanding of software complexity. For our evaluation using Weyuker's properties, we use the language defined in the original paper.

Several sources in the literature have remarked —in reference to Zuse [17]— that Weyuker's Properties suffer from a logical error and cannot be completely fulfilled by a single measure (e.g. [4, 11]). Zuse presumably uses a different set of properties that were perhaps defined in an older publication

ICPC'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...$15.00
http://dx.doi.org/10.1145/2597008.2597794

by Weyuker, which is not archived and thus unavailable. Weyuker herself makes no mention of this earlier set of properties. Part of the criticism by Zuse focused on a combination of properties that is not present in Weyuker's publication from 1988 [16].

DepDegree. DepDegree is a measure for data-flow dependencies that counts the dependencies for each statement in the use-def graph [1]. The DepDegree for a method is the sum of the dependency-counts for its statements. It is based on the psychological insight that the number of items that can be held in immediate memory is limited (to seven chunks, plus or minus two) [10]. Therefore, code that requires the programmer to remember many data-flow facts concurrently is hard to understand and difficult to maintain.

3. EVALUATION OF DEPDEGREE

The contribution of this paper is to show that all nine of Weyuker's properties hold for DepDegree. For more information, we refer to the supplementary web page [1].

THEOREM: DepDegree satisfies Properties 1 to 9.

PROOF: We prove each property separately in the following. Most properties are existential, such that we give a witness for the properties. Notation: for programs P and Q, we write $P \equiv Q$ if P and Q are functionally equivalent (same input/output relation, program variables, termination [16]), and $|P|$ for the measurement value of a program P.

Property 1. There exist two bodies P and Q with $|P| \neq |Q|$.

This property requires that the measure maps to at least two different measurement values. This property ensures that the measure not simply maps all values to one result. This is trivially fulfilled for DepDegree.

Property 2. For a given non-negative number c, there are only finitely many programs with measurement value c, provided that (1) the number of identifiers is finite, (2) there exists a largest possible number that can be represented in the program, (3) there is an upper bound on the length of an instruction, and (4) every reasonable counterexample should contain intra-block data flow.

This strengthening of Property 1 ensures that not too many programs are mapped to the same measurement value. As such, it excludes classification functions that map programs to a small set of values. This property ensure a certain distinguishing power between measurement values.

Several statements do not increase the DepDegree of a program body. Every predicate or expression that is composed only of constants yields a DepDegree of 0. An assignment of an expression with DepDegree 0 has itself a DepDegree of 0. Similarly, a loop or condition with a predicate with DepDegree value 0 and bodies consisting only of statements with a DepDegree of 0 can be constructed. The only way of constructing an infinite number of programs with the same DepDegree is to use variations and permutations of such programs. However, no intra-block data flow occurs in examples constructed this way, violating the fourth requirement. Every counterexample based on data flow inevitably increases the DepDegree of the body. Therefore, for every DepDegree value c, there exist only finitely many programs that fulfill the requirements. As a result, DepDegree fulfills Property 2.

[1] http://www.sosy-lab.org/~dbeyer/DepDegreeProperties/

Property 3. There are distinct programs P and Q such that $|P| = |Q|$.

```
1 PROGRAM(a)
2   IF a = 0 THEN
3     out ← a
4   ELSE
5     out ← a + 10
6   END
7 OUTPUT(out)
```

```
1 PROGRAM(a)
2   b ← a
3   c ← b + 2
4   d ← c * 2
5   out ← d − 4
6 OUTPUT(out)
```

Figure 1: Programs P and Q

To ensure that not every program maps to a distinct value, this property requires that there are at least two different programs that yield the same measurement value.

Two programs have the same DepDegree value if the number of data-flow dependencies is equal. A non-trivial example where DepDegree satisfies this property is given in Fig. 1: it shows two distinct programs with DepDegree value 5; the first program uses a conditional assignment and the second a sequence of assignments.

Property 4. There exist two programs P and Q such that $P \equiv Q$ and $|P| \neq |Q|$ hold.

```
1  PROGRAM(a, b, c)
2    IF a ≤ b THEN
3      IF a ≤ c THEN
4        min ← a
5      ELSE
6        min ← c
7      END
8    ELSE
9      IF b ≤ c THEN
10       min ← b
11     ELSE
12       min ← c
13     END
14   END
15 OUTPUT(min)
```

```
1 PROGRAM(a, b, c)
2   min ← a
3   IF b < min THEN
4     min ← b
5   END
6   IF c < min THEN
7     min ← c
8   END
9 OUTPUT(min)
```

Figure 2: Programs P and Q

Different ways to express the same functionality exist, some more complex than others. A measure for complexity should not only quantify the relation of input to output, but be sensitive to the details of the implementation and thus distinguish between differently complex implementations. Thus, this property requires that at least two different programs with equal functionality yield distinct measurement values.

To show this property, consider the two implementations of a function calculating the minimum of three values in Fig. 2. For any assignment of variables a, b, and c, both P and Q produce the same output value $min = \min(a, b, c)$. Counting the number of dependencies shows that $|P| = 14$ and $|Q| = 11$. Therefore, $P \equiv Q$ and $|P| \neq |Q|$ holds and we conclude that DepDegree fulfills Property 4.

Property 5. For all program bodies P and Q, the conditions $|P| \leq |P;Q|$ and $|Q| \leq |P;Q|$ hold.

This property of "monotonicity" requires that the measurement value only grows and never decreases when composing a program body by adding another block before or after the original block.

To show that this property holds, we have to consider what happens in all possible compositions of program bodies. The DepDegree of a body is determined by the number of definitions of variables that are used in program operations, either on the right hand side of an assignment or as part of a condition. As by the requirement of the language, we know that every variable must be defined prior to being used, making the DepDegree for every variable usage at least 1.

In a composition, a program body P prior to a program body Q can have the following effect on the measurement value of DepDegree: If there are no assignments in P that are used in Q, $|P;Q|$ is at least that of $|Q|$. If there is an assignment in P that is used in Q, the DepDegree of the

usage of this variable in Q is at least 1, depending on the nature of the assignment in P. Thus, we have $|Q| \leq |P;Q|$.

Consider the DepDegree of a program body P followed by a program body Q. Again, it depends on the number of variable definitions in P and usages in Q. Adding a program body Q after P can only increase the number of dependencies in the new composite body and thus leads to an equal or higher DepDegree. Thus we have $|P| \leq |P;Q|$.

Given that both above requirements hold for two bodies P and Q, the conjunction of the two requirements holds as well and DepDegree fulfills Property 5.

Property 6.

(a) There exist program bodies P, Q, and R_a such that $|P| = |Q|$ and $|P;R_a| \neq |Q;R_a|$ hold.

(b) There exist program bodies P, Q, and R_b such that $|P| = |Q|$ and $|R_b;P| \neq |R_b;Q|$ hold.

```
1 a ← 1
2 IF a < b THEN
3   a ← b
4 END
```

```
1 IF a < c THEN
2   c ← 0
3 END
4 a ← b
```

```
1 d ← a
```

```
1 IF c < 10 THEN
2   b ← c
3 ELSE
4   b ← 10
5 END
```

Figure 3: Program bodies P, Q, R_a, and R_b

A program body may interact differently with two program bodies of equal measurement values. A measure is required to be responsive to this kind of interaction. Thus, measures that simply count statements, without considering interactions, do not fulfill this property. For case (a), a program body is attached after two different program bodies with equal measurement values. For a measure to fulfill this criterion, the composition $|P;R_a|$ must yield a value different from $|Q;R_a|$. For case (b), the body R_b is added to the front of the bodies of P and Q.

Consider the program bodies P and Q in Fig. 3, both with a DepDegree value of 3. Because the composition $|P;R_a|$ yields 5, whereas $|Q;R_a|$ yields 4, Prop. 6a holds. In the program body $R_b;P$, composed from R_b and P, we obtain a DepDegree of 7. In contrast to that, the measurement value of DepDegree is 6 for the composed body $R_b;Q$. Thus, the requirement for Prop. 6b is also fulfilled, and we conclude that DepDegree fulfills Prop. 6.

Property 7. There are programs P and Q such that Q is obtained from P by permuting the order of body parts of P, and $|P| \neq |Q|$.

```
1 PROGRAM(a, b)
2   a ← a + b
3   IF 10 < a THEN
4     a ← 10
5   END
6 OUTPUT()
```

```
1 PROGRAM(a, b)
2   IF 10 < a THEN
3     a ← 10
4   END
5   a ← a + b
6 OUTPUT()
```

Figure 4: Programs P and Q

The intention is to ensure that not only the presence of operations but also their potential interaction is evaluated by the measure. This is especially relevant because often the order of statements defines the outcome. Similar to Prop. 6, this property ensures that interactions are considered by the measure, but here on statement level, whereas Prop. 6 makes a statement on larger blocks.

Fig. 4 is an example of a permutation that fulfills this property for DepDegree. In both programs, the condition of the if-statement has a DepDegree value of 1. But whereas the assignment in P in line 2 has only one dependency for variable a, the variable a used in the assignment in Q in line 5 is defined both in lines 1 and 3 and has therefore two dependencies. This yields $|P| = 3 \neq 4 = |Q|$, and thus, Property 7 is satisfied by DepDegree.

Property 8. If P is a renaming of Q, then $|P| = |Q|$.

The quality of names for variables, methods, and classes play an important role in the understandability of software systems. While a poorly chosen name may give no or even confusing information to a programmer, an artfully crafted identifier informs the reader about the intention behind a variable, method, or class. But as program documentation it is part of human interaction. Because this information cannot be quantified by a program-based measure, an evaluation of variable names must not be part of a structural analysis.

DepDegree does not evaluate variable names for the calculation of the measurement result. Subsequently, this property also holds for DepDegree.

Property 9. There exist program bodies P and Q such that $|P| + |Q| < |P;Q|$ holds.

```
1 a ← 1
2 IF b = 0 THEN
3   a ← 0
4 END
```

```
1 c ← 1
2 IF a = 0 THEN
3   c ← 0
4 END
```

Figure 5: Program bodies P and Q

To ensure that a measure reflects a possible increase in complexity if an interaction of two concatenated bodies occurs, this property requires the existence of two program bodies whose concatenation has a measurement value that is greater than the sum of its parts.

If considering either of the program bodies P and Q in Fig. 5 on its own, their respective DepDegree is 1. But if P and Q are concatenated to form $P;Q$, the condition in line 2 of Q has a DepDegree of 2, because it depends on both the definitions in lines 1 and 3 in P. $|P;Q|$ is therefore 3, which is larger than $|P| + |Q| = 2$. This establishes that DepDegree also satisfies this property.

4. DISCUSSION

We have shown that DepDegree fulfills all nine properties proposed by Weyuker. This establishes the formal validity, according to Weyuker, for using DepDegree as an indicator for program complexity. Table 1 gives an overview, including results by Weyuker [16] and Misra [12]. The symbols ✓ and ✗ indicate that a property is fulfilled or not fulfilled, respectively.

Cyclomatic complexity measures the number of linearly independent paths in the control-flow graph of a method and is often used as an indicator for procedure complexity [9]. The placement of statements is not considered, and therefore, Properties 2, 6, 7, and 9 are not fulfilled.

Statement count is a widely used measure of program size. It does not satisfy Properties 6, 7, and 9, because interactions between statements are not taken into account.

Data-flow complexity measures the number of edges between blocks in the use-def graph of a method [14, 16]. The measure is based on the idea that the use-def relationship between variables is easier to understand if the definition and usage of a variable are inside the same block. The measure does not consider all edges of the use-def graph, instead only those that contribute to data-flow between blocks. As such, the measure 'data-flow complexity' does not meet the criteria needed to fulfill Properties 2 and 5.

Table 1: Complexity properties fulfilled by the measures

Property	Cyclomatic Complexity	Statement Count	Data-Flow Complexity	Halstead Effort	Cognitive Complexity	DepDegree
1	✓	✓	✓	✓	✓	✓
2	✗	✓	✗	✓	✓	✓
3	✓	✓	✓	✓	✓	✓
4	✓	✓	✓	✓	✓	✓
5	✓	✓	✗	✗	✓	✓
6	✗	✗	✓	✓	✗	✓
7	✗	✗	✓	✗	✓	✓
8	✓	✓	✓	✓	✓	✓
9	✗	✗	✓	✓	✓	✓

Halstead effort is part of Halstead's suite of software measures [7]. The measure tries to represent the effort that is necessary to implement a program and is defined as $e = \frac{V^2}{V^*}$, where V is the volume of a program (a measure of program size) and V^* is the minimum possible length of the implementation of a program (in its original definition, V^* is not computable; adoptions are necessary). Property 5 is not fulfilled by Halstead effort, because adding additional statements will not necessarily increase the measurement result and may even reduce it. Also Property 7 does not hold, because the sequence of statements and their interaction is not taken into account.

The *cognitive complexity* is a measure that is defined on the interactions of basic control structures and is an indicator for the complexity of the logical structure of the software [12]. It intends to represent the time and effort needed to comprehend source code. This measure does not fulfill Property 6, because concatenation of two program bodies does not necessarily increase the measurement result.

DepDegree is the only measure so far that fulfills all of Weyuker's properties, as shown in Sect. 3 and summarized in Table 1. This formal evaluation increases the confidence in DepDegree as an indicator for program complexity.

5. CONCLUSION

We investigated the possibility to use the data-flow-based measure DepDegree [1] as indicator for program complexity, instead of the well-known, classic measures by McCabe, Halstead, and others. The idea behind DepDegree is that a program is the more difficult to understand the more chunks the programmer's short-term memory has to remember [10]. This approach is new and promising, but was not yet thoroughly investigated. Our work is part of a larger project that includes controlled experiments on software comprehension.

We present the results of our formal evaluation of the measure DepDegree. So far, the argument in the community was that Weyuker's properties are too strict and no measure can fulfill them. This situation has changed and the argument is no longer valid: in this paper we establish —for the first time in the literature— that a reasonable indicator for complexity of program code exists and in fact fulfills all of Weyuker's properties. DepDegree not only fulfills all nine properties, but is even a technically simple and easy to understand measure, which is easy to integrate into measurement frameworks.

Weyuker's properties represent reasonable requirements that an indicator for complexity of program code should satisfy. It is surprising that measures that do not fulfill the required properties (cf. Sect. 4) are widely used in practice. Software-engineering researchers should more emphasize on designing measures that fulfill the properties. We have shown that this is possible — thus, the relevance of measures that do not fulfill Weyuker's properties should be re-evaluated.

This paper provides a formal validation of the measure DepDegree. Empirical evaluations of software measures are not part of this work. Prior research established that DepDegree reflects programmer's opinion better than the cyclomatic complexity [8]. These results, together with our formal validation, declare DepDegree as a measure with high potential for practical use and that empirical evaluations should establish further results about this measure. Formal evaluations are a minimal requirement for providing practitioners and researchers the necessary confidence in the validity and soundness of measurement results (cf. also [4, 16]).

6. REFERENCES

[1] D. Beyer and A. Fararooy. A simple and effective measure for complex low-level dependencies. In *Proc. ICPC*, pages 80–83. IEEE, 2010.

[2] J. Cardoso. Evaluating the process control-flow complexity measure. In *ICWS*, pages 803–804, 2005.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[4] N. E. Fenton. Software Measurement: A Necessary Scientific Basis. *IEEE Trans. Software Eng.*, 20(3):199–206, 1994.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[6] Gursaran and G. Roy. On the applicability of Weyuker property 9 to object-oriented structural inheritance complexity metrics. *IEEE Trans. Software Eng.*, 27(4):381–384, 2001.

[7] M. H. Halstead. *Elements of Software Science (Operat. and Progr. Sys. Series).* Elsevier, 1977.

[8] B. Katzmarski and R. Koschke. Program complexity metrics and programmer opinions. In *Proc. ICPC*, pages 17–26. IEEE, 2012.

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

[10] 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.

[11] S. Misra. Modified Set of Weyuker's Properties. In *Proc. ICCI*, pages 242–247, 2006.

[12] S. Misra and A. K. Misra. Evaluating cognitive complexity measure with Weyuker properties. In *Proc. ICCI*, pages 103–108, 2004.

[13] S. Misra and A. K. Misra. Evaluation and comparison of cognitive complexity measure. *ACM SIGSOFT Software Engineering Notes*, 32(2):1–5, 2007.

[14] E. I. Oviedo. Control flow, data flow and program complexity. In *Proc. COMPSAC*, pages 146–152, 1980.

[15] A. Sharma, R. K. Bhatia, and P. S. Grover. Empirical evaluation and validation of interface complexity metrics for software components. *Int. J. Software Eng. and Knowledge Eng.*, 18(7):919–931, 2008.

[16] E.J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14:1357–, 1988.

[17] H. Zuse. *Software Complexity: Measures and Methods.* deGruyter, 1991.