6 ||

Calculating Software Complexity Using the Halting Problem

Jonathan Bartlett

The Blyth Institute

Abstract

Calculating the complexity of software projects is important to software engineering as it helps in estimating the likely locations of bugs as well as the number of resources required to modify certain program areas. Cyclomatic complexity is one of the primary estimators of software complexity which operates by counted branch points in software code. However, cyclomatic complexity assumes that all branch points are equally complex. Some types of branch points require more creativity and foresight to understand and program correctly than others. Specifically, when knowledge of the behavior of a loop or recursion requires solving a problem similar to the halting problem, that loop has intrinsically more complexity than other types of loops or conditions. Halting-problem-like problems can be detected by looking for loops whose termination conditions are not intrinsically bound in the looping construct. These types of loops are counted to find the program complexity. This metric is orthogonal to cyclomatic complexity (which remains useful) rather than as a substitute for it.

1 Complexity Metrics in Software

Managing software development is often about managing risks - knowing which tasks are likely to take more time than others, which features are more likely to impact others, how much testing will be required to make sure that a feature is solid, and whether a bug fix or a feature implementation requested right before release will be more likely to make the code more stable or lead to other bugs.

One of the key considerations of risk management is software complexity. Complex software is inherently more difficult to build, test, and maintain. Therefore, it is critical for software development managers to know which parts of code are most complex and therefore more likely to incur failures if modified.

2 A Brief History of Software Complexity Metrics

Early complexity metrics were based almost entirely on the amount of code produced. Therefore, a function which contained 10 lines of code was considered more complex than one which contained only 5. However, since "lines" of code often varies due to stylistic differences, Halstead developed a set of measures based on the number of operators and operands present within the code (Kearney et al., 1986).

Lines of code and related metrics are still often used for software effort estimation, but its use in analyzing code complexity has fallen away. It quickly became clear that not all code is the same, and some operators are inherently more complex than others. Specifically, the decision structure of the program was inherently more complex than the computations. Cyclomatic complexity was created to measure the size of the decision structure of the program (McCabe, 1976). This is done by creating a graph of all basic code blocks as nodes and then adding edges which show how control can move between them. The formula for calculating the complexity is:

$$E - N + T \tag{6.1}$$

E is the number of edges, N is the number of nodes, and T is the number of terminating nodes (entry and exit points - usually 2).

Cyclomatic complexity is extremely useful in determining how to test software. The cyclomatic complexity of a program is also the minimum number of tests needed to cover every control flow branch of a program. If the cyclomatic complexity of a program is 5, then at least 5 tests must be devised to test every branch of code. Such tests do not guarantee total coverage of all possible test conditions, but they will verify that every statement in the program will contribute to at least one test.

The ABC metric is a simplified metric combining aspects of both lines of code and cyclomatic complexity. It works by simply counting the number of assignment statements, the number of branches (direct flow control shifts - i.e., function calls), and the number of conditionals (Fitzpatrick, 1997, pp. 2–3). These can then be analyzed on a whole program, per-module, or per-function basis, to give an overview of complexity and size of a software program.

3 Deeper Difficulties in Software

While each of the previously-mentioned metrics have their usefulness, none of them get at the deeper difficulties that make software projects complex. Many program-

ming languages attempt to remove the inherent difficulties within software. Some, like COBOL, attempt to remove the mathematical notation common to programming languages. Others, like Java, try to simplify software by encapsulating related methods into objects. Visual programming languages, such as Flowcode, turn all software into visual representations, allowing the user to drag and drop flowchart-like components to accomplish programming tasks. The idea is that it is the text-based nature of the software which causes complexity and confusion within software.

While each of these actually do relieve certain specific problems in software development, none of them are able to remove the complexity of software development because the complexity is inherent in the nature of software development itself. What makes software development difficult is its open-ended nature. Most general-purpose programming languages today are universal in nature - that is, they can perform any computable function that any other programming language can perform. Thus, they are open-ended - the types of operations that they perform are entirely specifiable by the programmer and are not restricted. Universal programming languages are chaotic - that is, there is no easy mapping between programs, data, and results. Therefore, predicting the output over a wide swath of code and data can be difficult.

Languages are chaotic because of arbitrary looping structures. Interestingly, this is precisely the same part that causes it to be open-ended. Arbitrary looping structures allow a programmer to generate any possible computable function. As such, they also allow a programmer to write programs whose results are chaotic. In practical terms, arbitrary looping structures are `while` statements, `jump`/`goto` statements, recursive functions, and continuations, though others may be possible.

Occasionally, the solution to this problem has been to reduce the scope of the language. SuperGlue is one language which is specifically designed to be as expressive as possible while avoiding constructs that lead to complexity (McDirmid, 2006). However, ultimately, to get beyond the originally conceived computational bounds of the programming language, universality, as well as the complexity that goes with it, are required.

4 The Halting Problem as an Insight Problem

As discussed elsewhere in this volume, some problems are not amenable to analytical analysis and require insight in order to solve them (Bartlett, 2014; Holloway, 2014). Neither are such problems computable - no computer is capable of calculating these problems. One such problem that is relevant is the halting problem.

The halting problem states that, given a universal language, there is no program that can be written which will tell if any arbitrary program written in that language will ever complete (i.e., halt). This is not based on the size of the program code, but rather on the nature of the constructs available. This is not to say that one could not write a program to tell if certain subsets of programs written in that

language will halt, but there could not be a program to tell if any given program would halt.

What is intriguing about this, as noted by Bartlett (2014), is that computer programmers seem to be able to possess this power to some degree. Since the complexity of problems assigned to them are arbitrarily hard (i.e., management, not the programmer, often decides what must be done), and the reason that arbitrarily hard programs are possible is because of the open-ended nature of universal programming languages (the parts of the language which *create* the chaos are precisely the ones *required* for programs of arbitrary complexity), it can be said that human programmers are generally reliable halting problem solvers.

There are cases (usually coming from number theory) where it is not known whether or not programs (even very simple ones!) halt. These seem to indicate that there are different levels of difficulty and different levels of insight required to make determinations. Some might even use these cases to argue against the general ability of humans to reliably solve the halting problem. However, the advancement of science and mathematics actually depends on the ability of humans to be able to accomplish such tasks. In other words, if the ability of humans to figure out such problems is doubted, then the progress of science itself is brought into question. Should mathematicians stop looking for answers in number theory? Or is it better to assume that the proper insight will come one day? The ability of programmers to reliably solve halting problems in their daily work should lend hope to the mathematicians that someone will eventually be able to have the insight to solve their problems as well.

5 Using the Halting Problem to Measure Software Complexity

The trouble with insight problems is that they are not reliably solved by individuals. They are unreliably solvable - in other words, a solution is possible, but there is no analytic procedure to do so. Even worse, if programmers do not realize that they are looking at an insight problem, they might not know that special care must be taken.

So how is an insight problem in code recognized? Since the types of programming structures which make a program universal have already been determined (i.e., arbitrary looping structures), those structures in code can therefore be detected.

For most programming languages, the main structures which must be detected are the following:

1. Loops where the iterations are not implicit in the control structure (called *open-ended loops*)

2. Recursive functions (which are just another way of implementing open-ended loops)

For open-ended loops, consider the following two programs that each print out the square of every number in an array:

```
ary.each{|x|
  puts x * x
}
```

Figure 6.1: A program with an implicitly-terminating loop

```
i = 0
while(i < a.length) {
  puts x * x
  i = i + 1
}
```

Figure 6.2: A program with an arbitrary looping structure

The implementation in Figure 6.2 is more complex than the one in Figure 6.1, but not because of the size of the program. What makes it more complex is that it utilizes an arbitrary looping structure - the while statement. In Figure 6.1, the looping is inherently bound by the loop operator. In Figure 6.2, the programmer must specifically act to make the loop terminate appropriately. There is no way an `each` statement on its own will fail to halt. There are many ways in which a `while` statement can fail to halt. The termination is decoupled from the loop construct itself.

Therefore, as a first pass to measure software complexity, the programmer can simply count the number of open-ended looping structures (either as loops or recursive functions) which occur in the program.

6 Adding Axioms to Minimize Insight

However, as is obvious from Figure 6.2, there are many well-understood conventions which mitigate the complexity of certain kinds of open-ended looping structures. In that figure, for instance, the variable i starts at zero, monotonically increases, and then terminates at a predetermined stopping point, which will result in the loop's termination. Even in cases where this sequence of steps is not codified within the language using a special statement or procedure, it is a well-understood looping convention. If the convention is followed correctly, the loop will terminate.

Gregory Chaitin formalized this idea in his algorithmic information theory. He pointed out that while certain problems were unsolvable given a base set of axioms,

by incorporating additional axioms into the problem, solutions can be found. For instance, following Chaitin, if God were to tell us how many programs of size N halted, that information could be used to solve the halting problem for programs of size N (Chaitin, 1982).[1]

In the same way, when a convention for constraining repetition is discovered, it can be incorporated into a canon of axioms which are also known to halt. And thus it should be treated almost on the same level as a language construct which produces close-ended loops, because language constructs enforce the validity of the axiom structurally, while conventions require the programmer to manually follow the convention correctly.

This canon of axioms can be codified into an extensible static analysis tool to check program complexity. Such a tool could consist of the set of potential non-terminating constructs, as well as a "book of conventions" which are the known conventions for ensuring termination. The tool would then measure the potential number of non-terminating constructs which do not conform to a pattern in the "book of conventions."[2]

In addition to the constructs which can be statically analyzed, some conventions (often termed as "patterns") will not be easily amenable to inference by software. They should, however, at least be documented, and they can be manually marked or removed after the fact. However, if the construct is not amenable to static analysis, extra effort should be taken to review all implementations of the construct manually.

It should also be recognized that these axioms should be treated as first-class insights—that is, the "book of conventions" should be considered a set of valuable intellectual assets. As solutions to insight problems, the "book of conventions" is by definition a set of solutions which are not immediately obvious, and, therefore, if a convention is not recorded, and is therefore "lost," it could well be a permanent loss of insight for an organization.

---

[1]For an informal proof of this, consider that the issue that makes the halting problem difficult is that if a program is running indefinitely, it is impossible to tell whether or not it is just taking a long time and will finish eventually, or if it truly will never finish. However, if it is known that k programs of size N will halt, one can simply run all programs of size N simultaneously. As long as the number of programs that have finished is smaller than k, then there are some programs in that set which will halt. Once all k programs finish, then it is needless to continue to run the remaining programs since it was given that exactly k programs finish. Therefore, if the number of programs in a set which halt is known ahead of time, it is possible to determine the answer to all the halting problems in a finite length of time—the length of time will be the maximum runtime of the longest running halting program.

[2]A similar procedure was independently developed by Bringsjord et al. (2006), Hertel (2009), and Harland (2007), though differing in many aspects and applications. Their solutions were to categorize non-halters rather than halters, and to do it based on runtime patterns rather than a static analysis of structural patterns in the program. They identified well-known patterns, data-mined for others, and then used a symbolic induction prover to match potential programs with these patterns. In addition, their purpose was for answering questions about computer science theory (specifically, the busy beaver problem) rather than assessing program complexity.

7 Using the Metric

The ultimate goal of the metric is to reduce the complexity of the software to zero. When a pattern which solves a restricted subset of the halting problem is discovered, it can be incorporated as a new axiom into the "book of conventions." Therefore, if there are any areas in the program which are marked as being complex, that means that it is still not known if the program will even finish! If a developer has a new insight into why a certain section of code will finish, this should be documented in the "book of conventions." If programmers cannot state why they think that the program will finish, it should be reviewed or rewritten. If the code cannot be reworked and the program cannot be proven to terminate, it should be considered highly suspicious.

In addition, areas of code which are complex given the constructs, but found in the "book of conventions" should be flagged for a second-pass review to make sure the conventions were followed appropriately. Such sections should also be flagged for programmers making modifications to be sure that their modifications do not upset the assumptions of the conventions.

8 Further Considerations

While this metric is very useful, it is obviously not the last word on complexity metrics. It does not technically supersede the other complexity metrics mentioned. Counting and estimating lines of code are still useful planning tools. Cyclomatic complexity is still a useful test coverage tool. However, this metric can be useful in identifying programming patterns which are intrinsically problematic and help mitigate possible problems with documentation, code review, and testing.

For future development, similar ideas could be applied not just to the halting complexity, but also to the complexity that variables are derived from. When the value of a variable is determined by multiple loops, or conditions within loops, or other sorts of non-linear mechanisms, the value of variables can be chaotic, even when they are not themselves what determines if the problem halts. Extending these ideas to variable calculation could allow for an even more comprehensive look at where program complexity lies.

References

Bartlett, J. (2014). Using Turing oracles in cognitive models of problem-solving. In J. Bartlett, D. Halsmer, & M. R. Hall (Eds.), *Engineering and the ultimate* (pp. 99–122). Broken Arrow, OK: Blyth Institute Press.

Bringsjord, S., Kellett, O., Shilliday, A., Taylor, J., van Heuveln, B., Yang, Y., Baumes, J., & Ross, K. (2006). A new Gödelian argument for hypercomputing minds based on the busy beaver problem. *Applied Mathematics and Computations*, 176(2), 516–530. Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.5786&rep=rep1&type=pdf

Chaitin, G. (1982). Gödel's theorem and information. *International Journal of Theoretical Physics*, 21, 941–954. Available from http://www.cs.auckland.ac.nz/~chaitin/georgia.html

Fitzpatrick, J. (1997). Applying the ABC metric to C, C++, and Java. *C++ Report*. Available from http://www.softwarerenovation.com/ABCMetric.pdf

Harland, J. (2007). Analysis of busy beaver machines with inductive proofs. In J. Gudmundsson & B. Jay (Eds.), *Cats'07: Proceedings of the 13th Australasian symposium on theory of computing* (pp. 71–78).

Hertel, J. (2009). Computing the uncomputable rado sigma function: An automated, symbolic induction prover for nonhalting Turing machines. *The Mathematica Journal*, 11(2), 270–283. Available from http://www.mathematica-journal.com/issue/v11i2/contents/Hertel/Hertel.pdf

Holloway, E. (2014). Complex specified information (CSI) collecting. In J. Bartlett, D. Halsmer, & M. R. Hall (Eds.), *Engineering and the ultimate* (pp. 153–166). Broken Arrow, OK: Blyth Institute Press.

Kearney, J. K., Sedlmeyer, R. L., Thompson, W. B., Gray, M. A., & Adler, M. A. (1986). Software complexity measurement. *Communications of the ACM*, 29(11), 1044–1050. Available from http://sunnyday.mit.edu/16.355/kearney.pdf

McCabe, T. J. (1976). A complexity measure. In *Proceedings of the 2nd international conference on software engineering*. Available from http://www.literateprogramming.com/mccabe.pdf

McDirmid, S. (2006). Turing completeness considered harmful: Component programming with a simple language. Submitted for publication. Available from http://lampwww.epfl.ch/~mcdirmid/papers/mcdirmid06turing.pdf