SOFTWARE COMPLEXITY MEASUREMENT USING MULTIPLE CRITERIA

Bhaskar Raj Sinha, Pradip Peter Dey, Mohammad Amin and Hassan Badkoobehi
National University, School of Engineering, Technology, and Media
3678 Aero Court, San Diego, CA 92123, U.S.A.
{bsinha, pdey, mamin, hbadkoob}@nu.edu

ABSTRACT

One of the challenges in software development is measuring and estimating software complexity at various levels of development. It is difficult to manage a software project without having well-developed measuring techniques. The term ‘software complexity’ is used to indicate the testability, maintainability, readability, and/or comprehensibility of a program. In the past, complexity is often measured by code size (lines of code), but that does not provide a measure of how difficult it is to understand or code the program. This research describes a quantitative method to measure software complexity, thus enabling us to apply appropriate resources towards developing, testing and maintaining the software. Using code abstractions in statechart diagrams, adequate measurement of software complexity is demonstrated.

INTRODUCTION

It has been estimated that a major part of business expenditures in computing is the maintenance of software. Since billions of lines of code (LOC) are estimated to exist worldwide, the software maintenance represents an Information Systems (IS) activity of considerable importance. Resource allocation for maintaining software needs to be based on certain aspects of software. Many measures of software complexity have been proposed for this and various other reasons [1-6, 10-15]. Most of these, although giving a good representation of complexity, do not specify a procedure for easy measurement. The McCabe metric [12] defines and assesses code complexity by the number of control edges created by the code, providing an objective, high-level measure of a program's complexity. Halstead’s method [6, 11] is based on the mathematical relationships among the number of variables, the complexity of the code, and the type of programming statements. Other methods to measure software complexity include code coverage, which is often used in testing, and specification-based methods, such as function point method [1] and its variations. Another method, the task complexity model, establishes measures to capture the complexity in the input and output data, in data manipulation and in relationships between software components. Another method defines complexity by examining the cognitive weights of the control structures in the software. A Metrics Suite for Object Oriented Design was introduced by Chidamber and Kemerer [3] on metrics specifically for object oriented code. They introduce six OO complexity metrics; weighted methods per class, coupling between object classes, response for a class, number of children, depth of inheritance tree and lack of cohesion of methods. Other methods used to measure software complexity are the Branching complexity
(Sneed Metric), Data access complexity (Card Metric), Data complexity (Chapin Metric) [2], and
the Decisional complexity (McClure Metric) [13].

This research describes a quantitative method to measure software complexity combining
certain aspects of the function point method [3], LOC, coding complexity abstractions, and
statechart diagrams [6-7]. The role of iterative development of graphical user interfaces and use
of objects such as buttons were not considered in the original function point method [1].
Statechart diagrams [6-7] are usually developed before the implementation phase and therefore
the results of the complexity measure are known at an early stage of the development process
enabling us to apportion resources toward developing, testing, and maintaining software. The
statecharts are augmented with some basic building block coding abstractions in order to obtain a
comprehensive complexity value of the software system in combination with certain aspects of
the function point.

SOFTWARE COMPLEXITY MEASUREMENT MODEL

Statechart diagrams are software models that represent dynamic aspects of software [7-8].
Statechart diagrams can represent any arbitrary computational problem, because they are
equivalent to Turing Machines [9]. A statechart diagram can be augmented with complexity
values of code building block abstractions in order to obtain a numeric value for the complexity
of the entire software system. Flow graph techniques enable us in quantifying fundamental
properties of software. Basic programming primitives are analyzed for their execution paths.
These building blocks may be combined to determine the number of execution paths in a piece of
code.

**Sequential code:** Figure 1 shows the flow graph for a sequential code. Number of execution
paths is one.

If ( ) then … : Figure 2 shows the flow graph for an If ( ) then … construct. Number of
execution paths is two – one when the test fails and the other when the test is true.

If ( ) then … else … : Figure 3 shows the flow graph for an If ( ) then … else … construct.
Number of execution paths is two.

If ( ) and ( ) then … : Figure 4 shows the flow graph for an If ( ) and ( ) then … construct.
Number of execution paths is three.

If ( ) or ( ) then … : Figure 5 shows the flow graph for an If ( ) or ( ) then … construct. Number
of execution paths is three.

Do … While ( ): Figure 6 shows the flow graph for a Do … While ( ) construct. Number of
execution paths is two.
**While ( ) Do … :** Figure 7 shows the flow graph for a While ( ) Do … construct. Number of execution paths is two.

**For ( ):** Figure 8 shows the flow graph for a For ( ) construct. Number of execution paths is two.

**Switch ( ):** Figure 9 shows the flow graph for a Switch ( ) construct. Number of execution paths is N. This is determined from the number of cases handled by the switch statement.

Using the coding abstractions of these primitives, a statechart diagram is augmented by adding numeric values to the transitions of the diagram based on the path value as shown in Figure 10. Assume that a small software project started with the following initial description of requirements: *Develop a software system for computing the volume of two types of storage units: box-storage and cylinder-storage. Users should be able to enter inputs interactively using a Graphical User Interface (GUI).* After studying the requirements, software engineers would discover that the system has to be web-based and it should be available 24/7. Users should be able to access the software without any login ID. The software engineers would then prepare a Software Requirements Specification (SRS) document and then proceed with requirements reviews and software design. In the design phase, a statechart diagram would be developed which specifies different states of the software according to its dynamic aspects. Figure 10 shows
a statechart diagram augmented with complexity values obtained from program building blocks suggested above. In a statechart diagram, a state is shown with a rounded rectangle. A state is a collection of values. Change in values results in a new state. A transition is an event or operation that transforms one state to another. The initial state is indicated by a small solid circle and the final state by a bull's eye.

Figure 10: A State-Chart Diagram for computing volume of Box and Cylinder units

The complexity measure of the software, called the complexity index, is initially estimated at the design stage and then calculated at the release time after the implementation and testing. Table 1 shows the estimation at the design phase using a statechart diagram and some additional features of the software, some of which were suggested by function point research.

<table>
<thead>
<tr>
<th>Contributing Factors</th>
<th>Estimated Values for the example software</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Total number of execution paths of the augmented state chart</td>
<td>14</td>
<td>Adding the numbers on the edges of Figure 10</td>
</tr>
<tr>
<td>2 Total number of edges</td>
<td>8</td>
<td>See Figure 10</td>
</tr>
<tr>
<td>3 10 points for each nested loop</td>
<td>0</td>
<td>No nested loop</td>
</tr>
<tr>
<td>4 5 points for each object referenced</td>
<td>110</td>
<td>22 * 5</td>
</tr>
<tr>
<td>5 10 points for each data-structure</td>
<td>10</td>
<td>Vector in Java</td>
</tr>
<tr>
<td>6 10 points for each external file</td>
<td>0</td>
<td>For the current version</td>
</tr>
<tr>
<td>7 20 -100 points for GUI</td>
<td>20</td>
<td>For the current version</td>
</tr>
<tr>
<td>8 10 points for each interface with another system</td>
<td>0</td>
<td>For the current version</td>
</tr>
<tr>
<td>9 One point for each 20 Lines of Code</td>
<td>5</td>
<td>Estimated 100 LOC</td>
</tr>
<tr>
<td>10 10 points for providing security with login ID and password</td>
<td>0</td>
<td>For the current version</td>
</tr>
<tr>
<td>11 100 points for additional level of security such as dealing with credit cards, social security number etc.</td>
<td>0</td>
<td>For the current version</td>
</tr>
<tr>
<td>Complexity index</td>
<td>167</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Complexity Index Estimation at the design phase
The estimated complexity index of the software was 167 as shown in Table 1. The software needed an expanding data-structure such as a vector; therefore 10 points are allocated in item# 5 of Table 1. The effort for GUI was comparatively low as indicated in Figure 11; therefore 20 points were assigned for item#7. The software was implemented using Object Oriented programming techniques in Java. Figure 11 shows a screen-shot of the running program. After the development of the example software, the actual complexity index was calculated at the release time. The complexity index of the software is 173 as shown in Table 2 which is very close to the initial estimation shown in Table 1. There were 23 actual objects as compared to the estimated 22 objects such as button. The actual LOC was 121 compared to 100 estimated earlier.

![Figure 11: A screen-shot of the working software](image)

<table>
<thead>
<tr>
<th>Contributing Factors</th>
<th>Values for the example software</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Total number of execution paths of the augmented state chart</td>
<td>14</td>
<td>Adding the numbers on the edges of Figure 10</td>
</tr>
<tr>
<td>2 Total number of edges</td>
<td>8</td>
<td>See Figure 10</td>
</tr>
<tr>
<td>3 10 points for each nested loop</td>
<td>0</td>
<td>No nested loop</td>
</tr>
<tr>
<td>4 5 points for each object referenced</td>
<td>115</td>
<td>23 * 5 (for twenty-three objects)</td>
</tr>
<tr>
<td>5 10 points for each data-structure</td>
<td>10</td>
<td>Vector in Java</td>
</tr>
<tr>
<td>6 10 points for each external file</td>
<td>0</td>
<td>For the current version</td>
</tr>
<tr>
<td>7 20 -100 points for GUI</td>
<td>20</td>
<td>For the current version</td>
</tr>
<tr>
<td>8 10 points for each interface with another system</td>
<td>0</td>
<td>For the current version</td>
</tr>
<tr>
<td>9 One point for each 20 lines of code</td>
<td>6</td>
<td>For 121 LOC</td>
</tr>
<tr>
<td>10 10 points for providing security with login ID and password</td>
<td>0</td>
<td>For the current version</td>
</tr>
<tr>
<td>11 100 points for additional level of security such as dealing with credit cards, social security number etc.</td>
<td>0</td>
<td>For the current version</td>
</tr>
</tbody>
</table>

Table 2: Complexity Index Calculation
CONCLUSION

A measurement method for software complexity is described which was influenced by the function point method [1]. It is justified by formal reasoning of statechart diagrams combined with informal intuitive reasoning of code building block abstractions and GUI efforts. Comparison of estimated and actual values based on an example demonstrates the viability of the method. This study may be extended with other constructs, including unstructured logic, interrupt handlings, process creation, web programming and other decisions. This method may also be used to estimate the impact of software complexity on the costs of software maintenance projects in IS environments.

REFERENCES