An Object Oriented Runtime Complexity Metric based on Iterative Decision Points

Amr F. Desouky¹, Letha H. Etzkorn²
1 Computer Science Department, University of Alabama in Huntsville, Huntsville, AL, USA
2 Computer Science Department, University of Alabama in Huntsville, Huntsville, AL, USA

Abstract – Software metrics are used to measure the quality of a software system. Such metrics indicate the level of desired quality present in a system. However, software metrics have traditionally been captured at compile time, rendering useful results, but often times inexact, as the complete source code differs from the executing subset. For this reason, static metrics can fall short of measuring the true operational behavior of object oriented programs. In this paper, we present an investigation into the runtime boundary behavior of Rhino 1.7R4 – an open source implementation of JavaScript, in which we introduce a new runtime metric that measures the quality of complexity based on iterative decision points. We call this the “runtime boundary” as we are instead measuring object oriented quality at runtime; normal performance metrics collected at runtime are typically neither object oriented nor focused on quality. Finally, we validate the metric by comparing it to bug data.

Keywords: Object Oriented Runtime Metrics, Complexity Measurement, Object Behavior, and Software Engineering.

1 Introduction

Object oriented software metrics have traditionally analyzed the quality of software systems at compile time [5]. Static, compile time measurements must consider the entire source code, since it’s not known at compile time which sections of code will actually execute. Therefore, static metrics have some degree of inaccuracy. However, some previous work [1, 2, 3, 4] has proposed a shift from the compile time boundary to runtime, allowing software complexity to be measured solely on a program’s runtime behavior. This approach of measurement yields improved accuracy as non-executed code is ignored during metric computation. For instance, consider a metric which determines the quality of complexity based on the number of method invocations achieved per object. To compute such a metric outside the runtime boundary (that is, at compile time) will prove inadequate as the exact number of calls made to a given method cannot be fully determined at compile time, primarily since program execution typically relies on external arguments such as user input, which is often irregular. These Runtime boundary metrics are object oriented, which differentiates them from typical performance metrics which are largely not focused on objects. Also, they examine quality factors such as complexity (or cohesion) at runtime, whereas typical performance metrics clearly focus on performance.

In this paper, we propose a new object oriented runtime complexity metric based on iterative decision points. A decision point is a conditional expression which can alter the control flow of the program resulting in the execution of a particular branch – sequence of code, over another [1]. Iterative decision points on the other hand are control structures which execute a code fragment repeatedly based on a single decision point. Common examples of iterative decision points are for loops, do-while, and while loops. To the best of our knowledge, object oriented runtime complexity metrics based on iterative decision points have never been examined before.
The remainder of this paper is organized as follows: Section 2 describes background information and related work. Section 3 defines our runtime complexity metric. Section 4 outlines the experimental design and analyzes results compared to bug data. And finally, section 5 concludes the paper and outlines future work.

2 RELATED WORK

While a large contribution has been made to static metrics, a limited body of work has been conducted in the field of Object Oriented Runtime Metrics. Mitchell et. al. [3] investigate whether objects of a class exhibit different behavior at runtime from a coupling perspective. They introduce a runtime object-level coupling metric based on Chidamber and Kemerer’s widely accepted CBO metric [5]. The authors conclude objects of the same class at the runtime boundary do exhibit different behavior than static metrics.

Mitchell et. al. [4] measure the quality of a software design using runtime object oriented metrics. The authors show that although some degree of correlation exists between runtime metrics and static metrics, runtime metrics capture properties not found in static metrics.

Mathur et. al. [1, 2, 6] present runtime metrics based on (1) decision points and (2) memory occupied by an object at runtime; both provide quality measurements of complexity. The former of the two counts the number of decision points executed per object for all selection structures: if, if-else, if-else-if, and switch – as well as repetition structures: for, while, and do-while. However, each decision point is only counted once, irrespective of the number of iterations. Our proposed metric is different because we are considering the number of iterations per decision point as a complexity metric itself.

3 RESEARCH APPROACH

Chidamber and Kemerer [5] have defined complexity: “The complexity of the class relates to simplicity, in that the more complex the class, the less simple the class”. For instance, a class comprised of inheritance, control structures, boolean logic, and methods, is more complex than a simple hello world class with a single method. To expand on this, a class with $1+n$ looping iterations is intuitively more complex than a class with only $1$ looping iteration. The extra cycles require CPU overhead to fetch, decode, and execute all instructions inside the loop, as well as memory, cache and register resources. Moreover, each additional cycle carries the risk of impeding performance in the event of a branch misprediction, ultimately resulting in penalties i.e., lost execution time. We use this intuitive understanding in defining our runtime complexity metrics.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RuNFA</td>
<td>Runtime Number of Functions Accesses for all instances of a class. Object Instances that do not access a function are not considered.</td>
</tr>
<tr>
<td>RuNOI</td>
<td>Runtime Number of Object Instances per class</td>
</tr>
<tr>
<td>RuNLI</td>
<td>Runtime Number of Looping Iterations for all instances of a class</td>
</tr>
<tr>
<td>RuCIDp-A</td>
<td>Runtime Complexity based on Iterative Decision Points $RuCIDp - A = \frac{RuNLI}{RuNFA}$</td>
</tr>
<tr>
<td>RuCIDp-B</td>
<td>Runtime Complexity based on Iterative Decision Points $RuCIDp - B = \frac{RuNLI}{RuNOI}$</td>
</tr>
<tr>
<td>RuCIDp-C</td>
<td>Runtime Complexity based on Iterative Decision Points $RuCIDp - C = \frac{\ln(RuNLI)}{RuNFA}$</td>
</tr>
</tbody>
</table>

Table 1. Runtime Metric Definitions
Consider the following example:

```java
class Example {
    void funct_1(int n) {
        while (n < 10) {
            n++;
        }
    }
    void funct_2(int n) {
        for (int i = 0; i < n; n++) {
            continue;
        }
    }
    void funct_3(int n) {
        do {
            n++;
        } while (n < 10);
    }
    void funct_4(int n) {
        for (int i = 10; i > n; n--) {
            continue;
        }
    }
}
```

**Figure 1. Program Example**

Table 2 shows the runtime behavior of Figure 1 by assuming the number of looping iterations for a particular function of an object instance.

<table>
<thead>
<tr>
<th>Object Instances</th>
<th>funct_1</th>
<th>funct_2</th>
<th>funct_3</th>
<th>funct_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
<td>40</td>
<td>0</td>
<td>80</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>30</td>
</tr>
</tbody>
</table>

**Table 2. Runtime Results from Figure 1**

In reference to Table 2, we compute our metrics as follows:

<table>
<thead>
<tr>
<th>RuNFA</th>
<th>Count Object Instances (1, 3, 4, 5) = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RuNOI</td>
<td>Count Object Instances (1, 2, 3, 4, 5) = 5</td>
</tr>
</tbody>
</table>
| RuNLI | \[
\sum_{\text{iterations}} (\text{funct}_1, \text{funct}_2, \text{funct}_3, \text{funct}_4) = 10 + 20 + 2000 + 40 + 80 + 10 + 100 + 30 = 2290
\] |

**Table 3. Metric Computation Example**

\[
RuCIDp - A = \frac{RuNLI}{RuNFA} = \frac{2290}{4} = 572.5
\]

\[
RuCIDp - B = \frac{RuNLI}{RuNOI} = \frac{2290}{5} = 458
\]

\[
RuCIDp - C = \frac{\ln(RuNLI)}{RuNFA} = \frac{\ln(2290)}{4} = 0.83
\]

4 Experimental Study

In this section, we perform four case studies as a validation benchmark for our suggested metrics. For our case study, we used Rhino 1.7R4. The purpose of our validation is to determine whether RuCIDp-A, RuCIDp-B, and RuCIDp-C are good quality measures for object oriented complexity at runtime. We employ Pearson Product-Moment Correlation Coefficients to determine a correlation between the presence of bugs per class and our complexity metrics RuCIDp-A, RuCIDp-B, and RuCIDp-C. Our hypotheses for all three metrics are:

\[ H_0: \text{RuCIDp} \text{ has measurable impact in predicting the presence of bugs per class} \]

\[ H_1: \text{RuCIDp} \text{ has no measurable impact in predicting the presence of bugs per class} \]
4.1 Rhino

Rhino is an open source software package which serves as a JavaScript implementation written in Java. We selected a subset of 10 Rhino classes and modified them to compute our metrics. We chose these classes because they were the classes that mapped to bugs. Tags were applied to each repetition structure to track the Runtime Number of Iterations (RuNLi). In addition, each constructor was marked to track the Runtime Number of Object Instances (RuNOI) for a particular class. However, any object instance that did not access a loop was not counted. Finally, we tagged each function containing a loop to measure the Runtime Number of Functions Accessed (RuNFA).

4.2 Case Study 1

In our first case study, we analyze the presence of bugs and RuNLi. A normality test indicated the data was normal, so we employed Pearson’s correlation. The results of the Pearson’s correlation were not significant. However, an observation of the data set does show a number of bugs increasing with the number of loop iterations.

4.3 Case Study 2

In our second case study, we consider the correlation between the presence of bugs and RuCIDp-A. A test for normality shows RuCIDp-A data as not normal. The results of the Pearson's correlation were not significant. See Table 4.

4.4 Case Study 3

Our third study considered the correlation between the presence of bugs and RuCIDp - B. A test for normality shows data as not normal. The results of Pearson’s correlation were not significant. See Table 5.

4.5 Case Study 4

Our final case study considered the correlation between the presence of bugs and RuCIDp-C. We compute the numerator using a natural logarithm function in order to stabilize the variance sample of RuNLi because of the high iteration count. Log transformation is an accepted data transformation technique convenient for transforming extreme ranges into a normal distribution [9]. A test for normality shows RuCIDp-C data as normal. Thus, we employ Pearson Product Correlation. The results show a fairly large (according to the Hopkins scale) correlation of RuCIDp-C with bugs, while a measure of p-value also indicates a statistical significant correlation at the 90% confidence level.

<table>
<thead>
<tr>
<th>Pearson’s Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pearson</td>
</tr>
<tr>
<td>p-value</td>
</tr>
</tbody>
</table>

Table 4. Case 2 Pearson’s Correlation

<table>
<thead>
<tr>
<th>Pearson’s Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pearson</td>
</tr>
<tr>
<td>p-value</td>
</tr>
</tbody>
</table>

Table 5. Case 3 Pearson’s Correlation

<table>
<thead>
<tr>
<th>Pearson’s Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pearson</td>
</tr>
<tr>
<td>p-value</td>
</tr>
</tbody>
</table>

Table 6. Case 4 Pearson’s Correlation
5 Conclusions & Future Work

In this paper, we presented an experimental study into the runtime boundary behavior of Rhino 1.7R4 for computing our runtime metric. We observed a fairly large negative correlation and statistical significance at the 90% confidence level. The negative characteristic of the correlation was unexpected. However, we note that the correlation was relatively strong. We conjecture that perhaps software with a large number of loops receives extra attention from the programmer earlier on, and perhaps in some cases this could overcome problems related to any additional complexity through loop execution. Further study on different software packages is required.

This kind of situation would not have been seen in a static, compile-time examination of the program, because all loops would have been considered equal. Since our approach works dynamically, the execution of different loops could in fact be different.

Future work includes examining the runtime complexity behavior of self-iterative functions (i.e. recursion) and bugs using RuCIDp-C.

6 References


