On the costs of applying logic-based criteria to mobile applications:

An empirical analysis of predicates in real-world Objective-C and Swift applications

Juliana Botelho
Universidade Federal de Lavras
juliana.botelho@posgrad.ufla.br

Vinicius H. S. Durelli
Universidade Federal de São João del-Rei
durelli@ufsj.edu.br

Simone S. Borges
Universidade de São Paulo
sborges@icmc.usp.br

Andre T. Endo
Universidade Tecnológica Federal do Paraná
andreendo@utfpr.edu.br

Marcelo M. Eler
Universidade de São Paulo
marceloeler@usp.br

Marcio E. Delamaro
Universidade de São Paulo
delamaro@icmc.usp.br

Rafael S. Durelli
Universidade Federal de Lavras
rafael.durelli@dcc.ufla.br

ABSTRACT

The proliferation of mobile devices has given rise to an increasing demand for software that is well-suited to this particular environment. However, ensuring the quality of mobile applications is challenging. Much of the overall complexity of mobile applications stems from logic expressions, i.e., predicates, which appear in control-flow statements (e.g., `if`, `if-else`, `while`, and `do-while`) and define much of the behavior of software. Thus, testing predicates is key to ensuring the quality of mobile applications. We argue that an apt way to test predicates is by leveraging well-established logic-based criteria. Many logic-based criteria have been devised, e.g., the active clause coverage (ACC) and modified condition/decision coverage (MCDC). Given that ACC/MCDC are considered expensive, we set out to examine the cost of applying these criteria to mobile applications. We probed into a basic, but relevant, proxy for cost: the complexity of predicates, i.e., number of clauses in predicates. We examined 35 open-source mobile applications implemented in Objective-C and Swift ranging from 129 to 58,140 lines of code with a total of 19,345 predicates. We looked at the frequency and percentage of predicates. We also analyzed the relationship between overall measures of size and the frequency of predicates. We found that, although about 99% of the predicates in mobile application have at most three clauses, there is a significant positive linear correlation between overall measures of size and the number of predicates with four or more clauses. We conclude that mobile applications do not have many multi-clause predicates and hence sophisticated logic-based criteria are needed on only a small portion of the predicates.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SAST, September 18–19, 2017, Fortaleza, Brazil

© 2017 Association for Computing Machinery.

ACM ISBN 978-1-4503-5302-1/17/09...$15.00
https://doi.org/10.1145/3128473.3128477

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**;

KEYWORDS

Logic-based test criteria; Active clause coverage (ACC) criteria; Modified condition-decision coverage; Replication study

ACM Reference format:
Juliana Botelho, Vinicius H. S. Durelli, Simone S. Borges, Andre T. Endo, Marcelo M. Eler, Marcio E. Delamaro, and Rafael S. Durelli. 2017. On the costs of applying logic-based criteria to mobile applications:. In *Proceedings of SAST, Fortaleza, Brazil, September 18–19, 2017,* 9 pages.
https://doi.org/10.1145/3128473.3128477

1 INTRODUCTION

Logic predicates, which are expressions that evaluate to Boolean values, are common in a multitude of software artifacts. Although predicates appear in many software artifacts, this research focuses on predicates found in source code. Predicates are particularly common in source code, where they are instrumental in controlling the software's behavior and defining the possible flows of control. From a software testing standpoint, predicates are the most important part of control-flow statements (e.g., `if-else` and `while`) given that they play a central role in determining which blocks of code should be executed.

When predicates are wrong, the software behaves incorrectly. Given the importance and ubiquitousness of predicates in source code, testers devise test cases to ascertain whether

predicates behave as expected. This is called *logic-based testing* [1] and it is conducted during unit testing by designing tests from decisions in the source code.

Over the years, several logic-based criteria have been devised. The most straightforward logic-based criterion is concerned with designing test cases that cause predicates to become `true` and `false`. This criterion is often called *predicate coverage* (PC) [1].[1] It is also common to design tests that cause every portion (i.e., clause) of every predicate to evaluate to `true` and `false`, this is known as *clause coverage* (CC).[2] Two more rigorous criteria are *combinatorial coverage* (CoC)[3] and *active clause coverage* (ACC).[4] The former causes every predicate to take on all of its possible truth values. The latter has the goal of yielding test cases that cause every clause to become `true` and `false` while the other clauses have values that ensure that the clause under test dictates the outcome of the predicate [1].

The ever-increasing demand for mobile devices has given rise to a need for more and better software tailored to this particular environment. To provide the high-quality experience mobile users have come to expect, mobile applications have to be tested properly. Programmers try to create code that contains only simpler predicates, since it is easier to reason about these predicates. Nevertheless, effectively testing even simple predicates is hard. Hence, logic-based criteria are key to help testers face the complexities of mobile application testing. In other contexts, the importance of logic-based testing has already been recognized by several government agencies including the United States Federal Aviation Administration (FAA), which requires that ACC be used to certify safety critical parts of avionics software in commercial aircraft [12].

Since the cost-effectiveness of logic-based criteria hinges on the size of predicates, we set out to provide a greater understanding of the complexity of predicates found in mobile applications. To this end, we carried out an empirical study that examines the frequency and percentage of predicates found in mobile applications. We also looked at the correlation between overall measures of size (number of lines of code and source files) and the frequency of predicates.

Our motivation is based on the fact that if complex predicates are not common in mobile applications, sophisticated logic-based criteria are not needed. On the other hand, if complex predicates are common, high-end logic-based test criteria may be crucial to effective testing. We examined the predicates from 35 Objective-C and Swift programs and asked two questions:

(i) How many clauses do predicates that appear in real-world mobile applications have? *(ii)* Is the number of clauses per predicate correlated with the size of mobile applications, namely, are complex predicates more likely to appear in large mobile applications?

[1] PC is also called *decision coverage* (DC).
[2] CC is also known as *condition coverage*.
[3] CoC is also referred to as *modified condition coverage*.
[4] ACC is otherwise known as *modified condition, decision coverage* (MCDC).

Essentially, this is a replication study [14], repeating the empirical study conducted by Durelli et al. [7] but with programs implemented in different programming languages and that belong to different domains. The growing awareness of the importance of replication studies among software engineering researchers has lead them to realize that the true goal of empirical research is not conducting individual studies but developing an in-depth understanding of the benefits and shortcomings of various techniques. Oftentimes, it is hard to extrapolate the results of empirical studies to all possible domains, thus no single study on a technology or domain should be considered definitive [13]. As a result, many efforts have been made in conducting more replication studies in a variety of contexts: this paper falls into this category.

The remainder of this paper is organized as follows. Section 2 provides background on logic-based criteria, presenting definitions for the concepts in this paper. Section 3 describes the experimental design. Section 4 presents the results of the experiment and Section 5 describes our experimental results and the results of related work. Section 6 provides concluding remarks.

2 BACKGROUND

As mentioned, the result of a predicate is a Boolean value, i.e., either `true` or `false`. However, predicates may have Boolean and non-Boolean values. Relational operators ($>$, $<$, $\geq$, $\leq$, $=$, and $\neq$) are used to compare values within predicates and logical operators ($\wedge$, $\vee$, $\oplus$, $\rightarrow$, and $\leftrightarrow$) define the internal structure of predicates. The portions of a predicate, which are named *clauses*, are connected together by logical operators [1]. An example predicate is $(r \geq s) \wedge t$. This example contains two clauses: *(i)* a relational clause $(r \geq s)$, wherein r and s are non-Booleans, and *(ii)* a clause with only a Boolean value, t. One possible way to implement the example predicate in Objective-C is shown in Listing 1. The predicate appears inside an `if` statement and the two clauses are connected by the logical "and" operator (i.e., `&&`). The predicate accepts two integers and a Boolean value and, as mentioned, evaluates to a Boolean value.

Listing 1: Predicate containing two clauses implemented in Objective-C.

```
1  int r, s;
2  bool t;
3  ...
4  if (r >= s && t) { ... }
```

As previously stated, predicates and clauses are used in logic-based criteria, of which PC is the simplest example. Let P be a set of predicates with clauses C. For each predicate $p \in P$, C_p is the set of clauses in p. Formally, PC can be defined as follows [1, 7].

Definition 1 (PC). *For each $p \in P$, there are two test requirements: p evaluates to `true` and p evaluates to `false`.*

Considering again the example $(r \geq s) \wedge t$, two tests that satisfy PC are $\{r = 9, s = 3, t = \texttt{true}\}$ and $\{r = 5, s = 10, t = \texttt{true}\}$. It follows that PC requires two tests per predicate. The main shortcoming of this criterion is that it does not fully exercise individual clauses. For instance, in the example, t evaluates to `true` in both tests. CC remedies this problem by requiring tests at the clause level [1]:

Definition 2 (CC). *For each $c \in C$, there are two test requirements: c evaluates to* `true` *and c evaluates to* `false`*.*

For the example, CC can be satisfied by two tests: $\{r = 3, s = 6, t = \texttt{true}\}$ and $\{r = 6, s = 4, t = \texttt{false}\}$. CC also requires exactly two tests per predicate. Nevertheless, CC can be satisfied without causing the predicate to become both `true` and `false`, as in this example. The criterion that evaluates both individual clauses and the predicate by trying all combinations of values is CoC [1].

Definition 3 (CoC). *For each $p \in P$, the predicate must evaluate to each possible combination of truth values.*

Table 1 shows the complete truth table for $(r \geq s) \wedge t$. CoC requires 2^n tests, where n is the number of clauses. So CoC is unwieldy at best, being impractical for predicates with more than a few clauses [2, 7].

Table 1: Truth table for the predicate $(r \geq s) \wedge t$.

	$r \geq s$	t	$(r \geq s) \wedge t$
1	true	true	true
2	true	false	false
3	false	true	false
4	false	false	false

It is important to evaluate the overall effect of individual clauses on a predicate. One way to accomplish this is to evaluate predicates with all possible truth values. However, this entails creating 2^n tests (some of which might be infeasible), which is prohibitively expensive in most cases. To mitigate this issue, several logic-based criteria that exercise clauses with a reasonable amount of tests were devised. These criteria are based on the idea of making clauses "active" [1]. Active clause criteria are centered around the idea of *determination:* a clause is *active* (i.e., it independently determines the predicate's outcome) when changing its values changes the value of the predicate. Following the convention proposed by Offutt and Ammann [1], throughout this paper we refer to the clause under test as *major* clause, c_i. The other clauses, c_j, where $j \neq i$, are *minor* clauses. For ACC, each clause is considered in turn to be major [1, 7].

For the example $p = (r \geq s) \wedge t$ and assuming that $c_i = (r \geq s)$ is the major clause. It is easy to see that when the minor clause $(c_j = t)$ is `true`, the value of the predicate corresponds to the value of c_i. Hence, c_i determines p. ACC is based on selecting values for minor clauses so that the major clause determines the predicate. Each major clause results in two test requirements: c_i evaluates to `true` and c_i evaluates to `false`.

ACC yields four test requirements for our example predicate, two for clause $(r \geq s)$ and two for clause t. When $(r \geq s)$ is major: $((r \geq s) = \texttt{true}, t = \texttt{true})$ and $((r \geq s) = \texttt{false}, t = \texttt{true})$. The clause t, determines p if and only if $(r \geq s)$ evaluates to `true`. Thus, it entails two test requirements, $((r \geq s) = \texttt{true}, t = \texttt{true})$ and $((r \geq s) = \texttt{true}, t = \texttt{false})$. These test requirements are listed in the partial truth table in Table 2. The major clauses are shown in gray in Table 2.

Table 2: Partial truth table for $(r \geq s) \wedge t$.

	$r \geq s$	t	$(r \geq s) \wedge t$
1	true	true	true
2	false	true	false
3	true	true	true
4	true	false	false

It is worth noting that two test requirements are identical (the ones in rows one and three), so only three tests are needed. More specifically, ACC needs between $n + 1$ and $2n$ tests, where n represents the number of clauses. $n + 1$ is enough when $n < 4$, given the overlap among tests. The number of tests needed gets closer to $2n$ as n grows [7].

3 EXPERIMENT SETUP

This section outlines the setup we used to replicate the empirical study carried out by Durelli et al. [7]. In many ways, our setup is akin to the one described by Durelli et al. in [7]. We set out to examine the following research questions (RQs):

- **RQ_1:** How many clauses do most predicates in real-world mobile applications have?
- **RQ_2:** To what extent is the number of clauses per predicate related to overall measures of program size?

In our study, the size of mobile applications is determined by the number of physical lines of code (i.e., non-comment and non-blank) and the number of source files. In the next subsection we set forth the hypotheses we set out to examine.

3.1 Hypothesis Formulation

We formalized $\mathbf{RQ_2}$ into two hypotheses named *size influences predicates* (**SIP**):

- **Null hypothesis, SIP_0:** There is no association between the size of a mobile application and the number of clauses in its predicates.
- **Alternative hypothesis, SIP_1:** As the size of a mobile application increases, average predicate size also increases.

3.2 Variables Selection

Prior to carrying out experiments, conceptual definitions of the problem at hand need to be turned into measurable variables. This process is called *operationalization.* This subsection describes the *operational definitions* of the variables we set out to examine. The independent variables are *(i)* the number of physical lines of code and *(ii)* the number of source

files in each of the selected mobile applications. The number of predicates and their respective clauses are the dependent variables.

3.3 Goal Definition

Defining the scope of an experiment comes down to setting its goals. We used the organization proposed by the Goal/Question/Metric (GQM) [14] template to do so. According to this goal definition template, the scope of our study can be summarized as follows.

Object of study: The objects of study are open-source mobile applications implemented in Objective-C and Swift.

Purpose: The purpose of this experiment is to evaluate how cost effective logic-based criteria are by looking at how often predicates have at least four clauses. In addition, we also intend to examine if overall measures of size (i.e., number of lines of code and source files) exert any influence over the number of multi-clause predicates.

Quality focus: The primary effect under investigation is the average number of clauses per predicate. Given that criteria similar to ACC only have significant savings over CoC when predicates have more than three clauses, we are interested in determining how often predicates contain at least four clauses.

Perspective: from the standpoint of a researcher.

Context: We used 35 open-source mobile applications whose size ranges from 129 to 58,140 lines of code. These applications had from two to 475 source files.

3.4 Measurement Method

In order to analyze the predicates in open-source mobile applications we built upon and extended two ANTLR-based [11] grammars. We changed these grammars so the generated parsers would count the number of clauses in predicates implemented in Objective-C and Swift by examining control-flow statements.

3.5 Sample Selection

We randomly selected 35 mobile applications from GitHub [10], which is a web-based Git repository widely used to host open-source software projects.[5] More specifically, each mobile application in [10] was assigned a number and a simple random draw was made. Although the sample was randomly selected, we tried to balance the proportions of the selected applications by choosing applications that differed in size and complexity. When more than three applications that fall into the same category (e.g., web browsers) were randomly selected, only the first three applications were included into the final sample. The resulting sample contains web browsers (i.e., `Firefox` and `Onion Browser`), a text editor (i.e., `Edhita`), mobile games (i.e., `FlappySwift` and `Chess`), instant messaging applications (i.e., `Actor`, `Antidote`, and `Rocket.Chat`), an expense tracker application (i.e., `Buck Tracker`), photo editing applications (i.e., `Meme Maker` and `PixPic`), and a quick response (QR) code scanner (i.e., `QR Blank`). It is worth noting that all programs in our sample are real-world mobile applications.

[5] https://github.com/

4 EXPERIMENTAL RESULTS

The selected applications are listed in Table 3. Table 3 presents the number of physical lines of code (LOC), the number of source files (NSF), and the number of predicates in each of the mobile applications. It is worth emphasizing that code related to test suites was not taken into account in our analysis.

4.1 Analysis

The application with the most one-clause predicates was `Actor` (3,257) and the ones with the fewest one-clause predicates were `Colorblind` and `Chess`, with three one-clause predicates each. Several applications had predicates whose size ranged from six to 10 clauses: `Actor` (six), `rTracker` (six), `LogU` (four), `Wire` (three), `Buck Tracker` (two), `Concurrency` (two), `OnionBrowser` (two), `Money for GitHub` (one), `Dono` (one), and `Meme Maker` (one). Only three applications had predicates with more than 10 clauses: `Firefox for iOS`, `Actor`, and `rTracker`. `Firefox for iOS` was the only application with a predicate containing more than 15 clauses which was the largest predicate we found, containing 19 clauses. It can be seen that the amount of predicates in mobile applications steadily dwindles as the number of clauses increases, which is similar to the results reported by Durelli et al. [7] for Java and C/C++ applications.

Table 3 lists raw data on the mobile applications. Given that these applications vary in size, it is hard to pick out patterns. Thus, we calculated the percentage of different sized predicates in these applications. The resulting percentages are listed in Table 4.

The percentages in Table 4 would seem to suggest that one-clause predicates are by far the most common in all applications. The application containing the lowest percentage of one-clause predicates is `Chess` (60%), which is the second-smallest application containing 232 LOC. `Chess` is also the application with the highest percentage of two-clause predicates: 40%. `Game of Life` and `Alarm` contain the highest percentage of three-clause predicates: 10%. `Dono` contains the highest percentage of four-clause predicates: 0.966%. The applications with the highest percentage of predicates containing five clauses is `OnionBrowser` (0.288%). As shown in Table 4, on average, predicates whose size ranges from six to 10 clauses seem to be more common than five-clause predicates. The application with the highest percentage of predicates whose size ranges from six to 10 clauses is `rTracker` (0.339%, six predicates).

From analyzing Table 4, we see that, apart from `Workdays`, only application with more than 5,000 LOC have predicates with more than four clauses. Furthermore, only applications with more than 16,000 LOC have predicates whose size ranges

Table 3: Complexity of predicates in mobile applications. The entries in the table are in descending order by the number of lines of code (LOC).

Program Name	LOC†	NSF‡	Number of Predicates with n Clauses							
			n=1	n=2	n=3	n=4	n=5	n=6-10	n=11-15	n=16-20
`Wire`	58,145	475	2,929	331	41	18	4	3	0	0
`Firefox iOS`	51,214	349	1883	117	15	3	4	0	1	1
`Actor`	28,011	73	3,257	548	63	12	5	6	1	0
`LogU`	21,166	166	1,262	172	42	11	2	4	0	0
`rTracker`	16,433	54	1,598	142	15	6	3	6	1	0
`Buck Tracker`	15,187	122	1,005	178	30	7	3	2	0	0
`Monkey for GitHub`	11,756	91	1,047	78	11	5	0	1	0	0
`Antidote`	11,752	155	215	15	0	0	0	0	0	0
`Dono`	10,314	168	548	54	12	6	0	1	0	0
`Gulps`	9,521	127	237	19	0	1	0	0	0	0
`Meme Maker`	9,064	68	513	44	10	2	0	1	0	0
`Concurrency`	7,261	25	638	85	12	4	1	2	0	0
`OnionBrowser`	5,976	28	612	55	18	5	2	2	0	0
`PixPic`	5,290	83	232	10	1	0	0	0	0	0
`AlzPrevent`	4,531	75	126	5	2	0	0	0	0	0
`Speak`	3,421	31	91	6	0	0	0	0	0	0
`VPN On`	3,072	61	162	5	1	0	0	0	0	0
`Designer News`	2,009	31	120	4	0	0	0	0	0	0
`Workdays`	1,724	21	129	9	1	1	0	0	0	0
`Rocket.Chat`	1,296	44	31	1	0	0	0	0	0	0
`Swift 2048`	1,271	9	22	2	0	0	0	0	0	0
`Adblock Fast`	1,142	12	59	1	3	0	0	0	0	0
`HTY360Player`	1,054	6	79	0	0	0	0	0	0	0
`DownTube`	790	11	58	1	0	0	0	0	0	0
`Jim`	690	18	36	4	0	0	0	0	0	0
`OpenIt Today`	568	7	63	1	1	0	0	0	0	0
`Edhita`	552	8	26	1	0	0	0	0	0	0
`EMI Calculator`	386	10	4	1	0	0	0	0	0	0
`Colorblind`	353	12	3	0	0	0	0	0	0	0
`FlappySwift`	262	4	9	1	0	0	0	0	0	0
`Game of Life`	260	6	8	1	1	0	0	0	0	0
`Alarm`	254	8	9	0	1	0	0	0	0	0
`TODO`	238	6	15	0	0	0	0	0	0	0
`Chess`	232	14	3	2	0	0	0	0	0	0
`QR Blank`	129	2	5	0	0	0	0	0	0	0
Total	285,324	2,380	17,034	1,893	280	81	25	28	3	1
Max^{Raw}			3,257	548	63	18	5	6	1	1
Min^{Raw}			3	0	0	0	0	0	0	0
Mean			486.690	54.090	8	2.314	0.686	0.8	0.086	0.029
Percentages			88.05%	9.79%	1.45%	0.41%	0.13%	0.14%	0.02%	0.005%
			99.29%			0.68%			0.025%	

†Physical lines of code (non-comment and non-blank lines).

‡Number of source files (only Objective-C and Swift files were counted).

from 11 to 15 clauses. Some applications have a higher percentage of predicates with at least six and up to 10 clauses than five-clause predicates (e.g., `Actor`, `LogU`, and `rTracker`).

Table 4 also presents summary statistics over all subject programs. According to our results, the predicates in mobile applications have on average (mean) 90.99% one clause, about 7.35% contain two clauses, around 1.37% have three, only approximately 0.193% contain four clauses, only about 0.04% have five, around 0.06% contain six to 10 clauses, 0.004% contain 11 to 15 clauses, and only 0.001% have predicates whose size ranges from 16 to 20 clauses. This suggests that predicates with more than three clauses are rare.

Table 4: Percentage of n-clause predicates in mobile applications. The entries in this table follow the order in Table 3.

Program Name	Percentage of Predicates with n Clauses							
	n=1	n=2	n=3	n=4	n=5	n=6-10	n=11-15	n=16-20
Wire	88.063%	9.952%	1.233%	0.541%	0.120%	0.090%	0%	0%
Firefox iOS	93.034%	5.781%	0.741%	0.148%	0.198%	0%	0.049%	0.049%
Actor	83.684%	14.080%	1.619%	0.308%	0.128%	0.154%	0.026%	0%
LogU	84.528%	11.520%	2.813%	0.737%	0.134%	0.268%	0%	0%
rTracker	90.232%	8.018%	0.847%	0.339%	0.169%	0.339%	0.056%	0%
Buck Tracker	82.041%	14.531%	2.449%	0.571%	0.245%	0.163%	0%	0%
Monkey for GitHub	91.681%	6.830%	0.963%	0.438%	0%	0.088%	0%	0%
Antidote	93.478%	6.522%	0%	0%	0%	0%	0%	0%
Dono	88.245%	8.696%	1.932%	0.966%	0%	0.161%	0%	0%
Gulps	92.218%	7.393%	0%	0.389%	0%	0%	0%	0%
Meme Maker	90%	7.719%	1.754%	0.351%	0%	0.175%	0%	0%
Concurrency	85.984%	11.456%	1.617%	0.539%	0.135%	0.270%	0%	0%
OnionBrowser	88.184%	7.925%	2.594%	0.720%	0.288%	0.288%	0%	0%
PixPic	95.473%	4.115%	0.412%	0%	0%	0%	0%	0%
AlzPrevent	94.737%	3.759%	1.504%	0%	0%	0%	0%	0%
Speak	93.814%	6.186%	0%	0%	0%	0%	0%	0%
VPN On	96.429%	2.976%	0.595%	0%	0%	0%	0%	0%
Designer News	96.774%	3.226%	0%	0%	0%	0%	0%	0%
Workdays	92.143%	6.429%	0.714%	0.714%	0%	0%	0%	0%
Rocket.Chat	96.875%	3.125%	0%	0%	0%	0%	0%	0%
Swift 2048	91.667%	8.333%	0%	0%	0%	0%	0%	0%
Adblock Fast	93.651%	1.587%	4.762%	0%	0%	0%	0%	0%
HTY360Player	100%	0%	0%	0%	0%	0%	0%	0%
DownTube	98.305%	1.695%	0%	0%	0%	0%	0%	0%
Jim	90%	10%	0%	0%	0%	0%	0%	0%
OpenIt Today	96.923%	1.538%	1.538%	0%	0%	0%	0%	0%
Edhita	96.296%	3.704%	0%	0%	0%	0%	0%	0%
EMI Calculator	80%	20%	0%	0%	0%	0%	0%	0%
Colorblind	100%	0%	0%	0%	0%	0%	0%	0%
FlappySwift	90%	10%	0%	0%	0%	0%	0%	0%
Game of Life	80%	10%	10%	0%	0%	0%	0%	0%
Alarm	90%	0%	10%	0%	0%	0%	0%	0%
TODO	100%	0%	0%	0%	0%	0%	0%	0%
Chess	60%	40%	0%	0%	0%	0%	0%	0%
QR Blank	100%	0%	0%	0%	0%	0%	0%	0%
Max$^{\%}$	100	40	10	0.966	0.288	0.339	0.056	0.049
Min$^{\%}$	60	0	0	0	0	0	0	0
Mean	90.985	7.35	1.374	0.193	0.040	0.057	0.004	0.001
Std. dev.	7.733	7.380	2.417	0.287	0.081	0.102	0.013	0.008

Figure 1 outlines the data from Table 4 in boxplots. These boxplots clearly indicate that most predicates have less than four clauses. Predicates with more than four clauses are very rare in mobile applications. Based on our analysis of the data, we argue that the answer to $\mathbf{RQ}_1$ is that most predicates in mobile applications implemented in Objective-C and Swift have one, two, or up to three clauses. Very few predicates have four or more clauses.

4.1.1 Hypothesis Testing. We probed into the relationship between the measures of size describe in Section 3 and the frequency of predicates with at least four clauses. We started by generating a scatter diagrams (Figure 2) to test the hypotheses $\mathbf{SIP}_0$ and $\mathbf{SIP}_1$. The scatter diagrams show the lines of best fit for predicates whose sizes range from four to five clauses considering each independent variable: LOC and NSF. The linear regressions lines suggest that there is an association between the size of predicates and the size of mobile applications: that is, the size of the predicates and the size of mobile applications covary.

It is worth noting that the plots in Figure 2 do not fit a straight line, which indicates a departure from linearity. In addition, the boxplot in the bottom right corner of Figure 1

indicates that the data on four-clause predicates contain outliers. Therefore, we applied Spearman's rank correlation coefficient [8] to evaluate the correlation between predicate complexity and application size. Spearman's correlation coefficient measures the strength and direction of monotonic association between two variables. It is worth mentioning that monotonicity is not as restrictive as a linear relationship [3]. Correlation coefficient values (ρ) range from -1 to 1: 1 indicates a positive relation, -1 indicates a negative relation, and 0 represents no relation [8].

Table 5 lists the correlation coefficients between the percentage of predicates with n clauses (as shown in Table 4), and both the LOC and NSF. We interpret the correlation values in the following way: a large correlation value is 0.5 or greater, a medium correlation has a value of 0.3, and 0.1 indicates a small correlation [6].

The results in Table 5 suggest that there is a strong positive correlation between the percentage of predicates that contain four clauses or more and LOC and a medium positive correlation between these complex predicates and NSF. There is also a medium positive correlation between predicates with at least five and up to 15 clauses and LOC. As for NSF, this medium correlation is present only for predicates whose sizes range from six to 10 clauses.

The results in Table 5 provide us with enough evidence to reject the null hypothesis ($\mathbf{SIP}_0$). Our alternative hypothesis ($\mathbf{SIP}_1$) is partially supported by the results in Table 5: there is a strong correlation between predicates whose sizes range from five to 10 and LOC and NSF.

4.1.2 Threats to Validity. A threat to the external validity of our replication study is the representativeness of the subject programs. According to Miller [9], it is not possible to validate the representativeness of a given population. Despite the fact that all mobile applications used in our investigation are "real-world" programs, we cannot assure that we would reach similar conclusions if we had used a different sample. To mitigate this threat, we sought to eliminate selection bias by selecting the subjects randomly. We believe that our results are not far off in representing the population of mobile applications. Furthermore, to the best of our knowledge this is the largest study of predicates involving industrial-scale mobile applications.

As for the external validity of our study, it is worth mentioning that we are not sure whether our results can be generalized to other programming languages used to implement mobile applications. Thus, replications of this study are needed for different programming languages.

Potential threats to the construct validity stem from possible faults in the Objective-C and Swift parsers. We tried to mitigate these threats by evaluating the parsers against several small programs.

5 DISCUSSION AND COMPARISON WITH RELATED WORK

Our data indicate that most predicates in mobile applications are not very complex: most predicates have up to 3

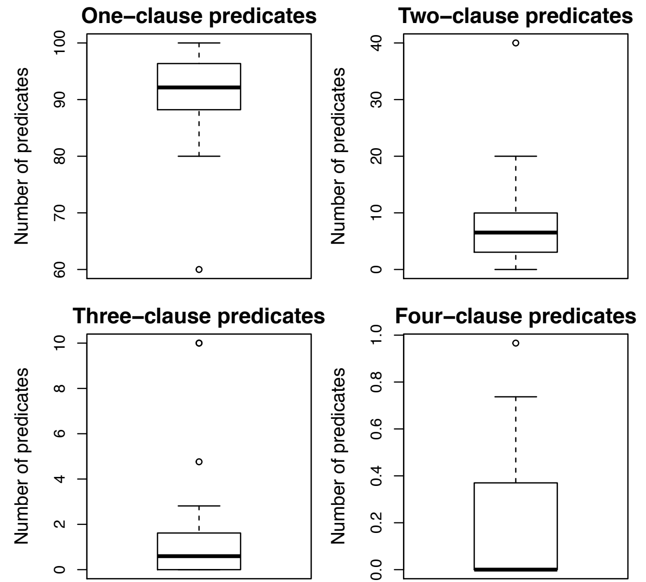

Figure 1: Distribution of predicates with up to four clauses. As indicated by the middle bar in the boxplots, which represents the median, four-clause predicates are uncommon.

clauses. From a testing standpoint, this can be seen as an indication that ACC is an interesting criterion to test mobile applications and not nearly as expensive as some might conjecture.

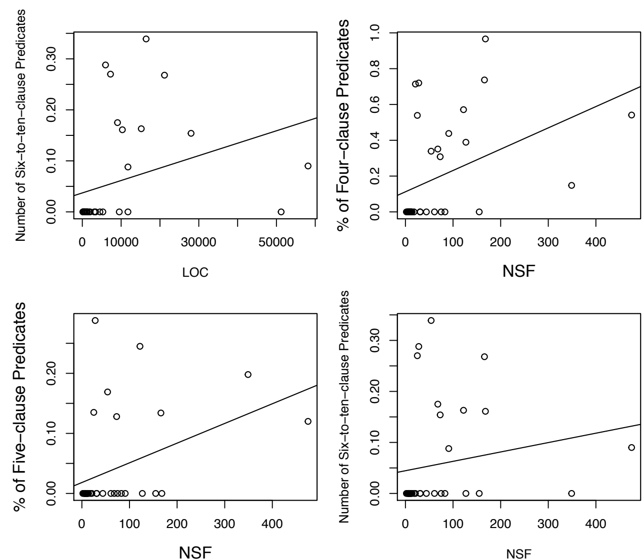

Figure 2: Scatter plots with regression lines that highlight the best linear approximation to the association between the complexity of predicates and measures of size.

As pointed out by Durelli et al. [7], there is a lack of studies investigating the complexity of predicates in "real"

software. In addition, previous studies were based on much smaller samples. Related work is listed in Table 6: the study by Durelli et al. [7], which investigated predicates in Java and C/C++ programs, a 2001 unpublished technical report, and a 1994 paper. Table 6 summarizes data from these studies and contrasts that data with data from our study. The top part of the table gives raw numbers of predicates whose sizes range from one, two, three, etc. The bottom part shows percentages of the total clauses.

Table 5: Correlation coefficients (ρ) indicating the extent to which the percentage of predicates with varying sizes is related to LOC and NSF (confidence level 95%).

	Spearman's Rank Correlation (ρ)	
Clauses	**LOC**	**NSF**
n=**1**	ρ = -0.294, p-value = 0.09	ρ = -0.277, p-value = 0.11
n=**2**	ρ = 0.369, p-value = 0.03	ρ = 0.341, p-value = 0.05
n=**3**	ρ = 0.381, p-value = 0.02	ρ = 0.287, p-value = 0.09
n=**4**	ρ = 0.717, p-value < 0.01	ρ = 0.615, p-value < 0.01
n=**5**	ρ = 0.637, p-value < 0.01	ρ = 0.437, p-value < 0.01
n=**6-10**	ρ = 0.652, p-value < 0.01	ρ = 0.472, p-value < 0.01
n=**11-15**	ρ = 0.443, p-value < 0.01	ρ = 0.271, p-value = 0.12
n=**16-20**	ρ = 0.272, p-value = 0.11	ρ = 0.272, p-value = 0.11

Our results confirm previous studies on the complexity of predicates. Predicates with one, two, or three clauses are by far the most common predicates (Table 6). Our results indicate that mobile applications are as complex as Java programs and not as complex as railway and airborne software: while 4.82% of the predicates in the safety-critical programs have four or more clauses, Java programs have only 0.67% predicates with at least four clauses and mobile applications implemented in Objective-C and Swift have only 0.71%.

6 CONCLUDING REMARKS

Predicates are key elements of control-flow statements, which are widely used in imperative programming languages. When predicates are erroneously encoded, the program's conditional behavior is bound to be wrong. Logic based criteria (e.g., PC, CoC, and ACC) can be used to test predicates.

As discussed, when predicates contain many clauses, PC is not effective and the cost of CoC becomes considerable. In effect, the motivation for using MCDC or ACC is based on the need to test large predicates. The results of our experiment show that mobile applications have few complex predicates. The main contribution of this research is that our results indicate that sophisticated logic-based criteria as MCDC and ACC are only needed on a small fraction of all the predicates. To the best of our knowledge [15], this is the first study to evaluate the cost of applying logic-based criteria to test mobile applications.

From a testing standpoint, the results of our study can be seen as an indication that ACC is an interesting criterion to test mobile applications and not nearly as expensive as some might conjecture. We elaborate on the cost of using logic-based criteria to test mobile applications in the next subsection.

6.1 Cost Analysis

Evaluating cost is notoriously complex because the notion of cost can be characterized and measured in many different ways. In this study, we characterize cost according to the three proxies used by Durelli et al. [7]:

(1) The number of tests needed. Tests add to the cost of testing since they need to implemented and run.
(2) Evaluating whether a set of tests satisfies a given criterion. This task can be carried out manually (2a), imposing human cost, or through tool support (2b), which imposes computational cost.
(3) Generating tests to satisfy a criterion. This task can also be carried either by hand (3a), which entails human cost, or automatically with a tool (3b), which imposes computational cost.

The number of complex predicates (i.e., multiclause predicates) mainly influences 1, 2a, and 3a. So our analysis emphasizes these costs.

We found that the vast majority of predicates have one clause (around 88%). For these predicates, PC, CoC, MCDC, and ACC have the same cost. That is, MCDC or ACC entail zero additional cost.

Two clause predicates are also prevalent in real-world mobile applications. PC yields two tests for two-clause predicates, MCDC and ACC require three, and CoC results in four tests. Considering the number of tests (proxy 1) for two-clause predicates, the cost of using MCDC and ACC is 50% higher than PC and 25% lower than CoC. As for 2a, (i.e., evaluating whether a test suite satisfies a given criterion by hand), the cost of CoC is less, but the same for cost 2b (i.e., using tool support to evaluate whether the tests satisfy the criterion).

According to our results, 1.45% of the predicates in mobile applications are three-clause predicates. These predicates require two tests for PC, four for MCDC and ACC, and eight for CoC. Considering the proxy for cost 1, the cost of using MCDC or ACC is 100% more than the cost of using PC, and 75% less than CoC. As for 2a, the cost of CoC is lower.

We found that 0.41% of the predicates have four clauses. For these predicates, two tests are needed for PC, five to eight for MCDC and ACC, and 16 for CoC. For proxy 1, the cost of using MCDC or ACC is approximately 250% to 400% more than PC and 50% less than CoC. Considering 2a, the cost of CoC is much less.

Table 6: Comparison with previous studies.

Source	Number of Predicates with n Clauses									
	n=1	n=2	n=3	n=4	n=5	n=6-10	n=11-15	n=16-20	$n >$=21	Total
Chilenski's report [4]	16,491	2,262	685	391	131	219	35	36	6	20,256
Software Tools [5]	446	72	9	0	0	0	0	0	0	527
Booch Components [5]	9,048	402	52	0	0	0	0	0	0	9,502
EFIS avionics display [5]	1,343	182	38	16	18	11	3	0	0	1,611
Durelli et al. [7] (Java)	354,660	38,048	5,414	1,647	465	497	53	21	6	400,811
Durelli et al. [7] (C/C++)	18,661	3,767	933	511	215	359	81	17	2	24,546
Our replication study	17,034	1,893	280	81	25	28	3	1	0	19,345

Source	Percentage of Predicates with n Clauses									
	n=1	n=2	n=3	n=4	n=5	n=6-10	n=11-15	n=16-20	$n >$=21	Total $n > 3$
Chilenski's report [4]	81.41%	11.17%	3.38%	1.93%	0.65%	1.08%	0.17%	0.18%	0.03%	4.04%
Software Tools [5]	84.63%	13.66%	1.71%	0%	0%	0%	0%	0%	0%	0%
Booch Components [5]	95.22%	4.23%	0.55%	0%	0%	0%	0%	0%	0%	0%
EFIS avionics display [5]	83.36%	11.30%	2.36%	0.99%	1.12%	0.68%	0.19%	0%	0%	2.98%
Durelli et al. [7] (Java)	88.49%	9.49%	1.35%	0.41%	0.12%	0.12%	0.013%	0.005%	0.0015%	0.67%
Durelli et al. [7] (C/C++)	76.02%	15.35%	3.80%	2.08%	0.88%	1.46%	0.33%	0.07%	0.008%	4.82%
Our replication study	88.05%	9.79%	1.45%	0.41%	0.13%	0.14%	0.02%	0.005%	0%	0.71%

Approximately 0.30% of the predicates have more than four clauses. These predicates need two tests for PC, at most $2n$ for MCDC and ACC (where n is the number of clauses), and 2^n for CoC. As for 1, the cost of using MCDC or ACC is $2n \times 100\%$: much less than CoC. The cost of CoC is much less considering 2a. An important result is that, similar to Java programs [7], the amount of predicates that have more than four clauses is negligible, which means that these extra costs are unlikely to be significant when considering the total development cost.

All in all, 3a (i.e., generating tests by hand to satisfy a given criterion) is much more expensive than 3b for all predicates with more than one clause for MCDC or ACC and CoC. Moreover, as noted by Durelli et al. [7], MCDC and ACC are harder than CoC for both 3a and 3b.

It is worth mentioning that the cost analysis we performed does not take into account a feasibility analysis. That is, some of the aforementioned test requirements may be infeasible. Consequently, when drawing conclusions from the above cost information, one should interpret it as a conservative bound on the number of test requirements of these logic-based criteria.

7 ACKNOWLEDGMENTS

Andre T. Endo is financially supported by CNPq/Brazil (grant number 445958/2014-6).

REFERENCES

[1] P. Ammann and J. Offutt. 2008. *Introduction to Software Testing.* Cambridge University Press. 344 pages.

[2] P. Ammann, J. Offutt, and H. Huang. 2003. Coverage Criteria for Logical Expressions. In *14th International Symposium on Software Reliability Engineering (ISSRE)*. 99–107.

[3] Bruce J. Chalmer. 1986. *Understanding Statistics.* CRC Press. 448 pages.

[4] J. J. Chilenski. 2001. *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion.* Technical Report DOT/FAA/AR-01/18. FAA Tech Center Report.

[5] J. J. Chilenski and S. P. Miller. 1994. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal* 9, 5 (1994), 193–200.

[6] J. Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences* (2nd ed.). Routledge Academic. 567 pages.

[7] V. H. S. Durelli, J. Offutt, N. Li, M. E. Delamaro, J. Guo, Z. Shi, and X. Ai. 2016. What to Expect of Predicates: An Empirical Analysis of Predicates in Real World Programs. *Journal of Systems and Software* 113 (2016), 324 – 336.

[8] J. Miles and M. Shevlin. 2000. *Applying Regression and Correlation: A Guide for Students and Researchers.* Sage Publications. 274 pages.

[9] J. Miller. 2004. Statistical Significance Testing - A Panacea for Software Technology Experiments? *Journal of Systems and Software* 73, 2 (2004), 183–192.

[10] Open-Source iOS Apps. 2017. https://github.com/dkhamsing/-open-source-ios-apps. (2017). Accessed: May 20, 2017.

[11] T. Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf. 328 pages.

[12] RTCA-DO-178B. 1992. Software Considerations in Airborne Systems and Equipment Certification. (December 1992).

[13] F. Q. B. Silva, M. Suassuna, A. C. C. França, A. M. Grubb, T. B. Gouveia, C. V. F. Monteiro, and I. E. dos Santos. 2014. Replication of Empirical Studies in Software Engineering Research: A Systematic Mapping Study. *Empirical Software Engineering* 19, 3 (2014), 501–557.

[14] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering.* Springer. 236 pages.

[15] S. Zein, N. Salleh, and J. Grundy. 2016. A Systematic Mapping Study of Mobile Application Testing Techniques. *Journal of Systems and Software* 117 (2016), 334–356.