The Journal of Systems and Software 113 (2016) 324–336

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

What to expect of predicates: An empirical analysis of predicates in real world programs

Vinicius H.S. Durelli[a,d,*], Jeff Offutt[b], Nan Li[b], Marcio E. Delamaro[a], Jin Guo[c], Zengshu Shi[c], Xinge Ai[c]

[a] Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, Brazil
[b] Software Engineering, George Mason University, Fairfax, VA, USA
[c] School of Information Science and Technology, Southwest Jiaotong University, Chengdu, Sichuan, China
[d] Faculdade Campo Limpo Paulista, São Paulo, Brasil

ARTICLE INFO

Article history:
Received 15 April 2013
Revised 1 October 2015
Accepted 14 December 2015
Available online 21 December 2015

Keywords:
Logic-based test criteria
Active clause coverage (ACC) criteria
Modified condition-decision coverage (MCDC) test criterion

ABSTRACT

One source of complexity in programs is logic expressions, i.e., predicates. Predicates define much of the functional behavior of the software. Many logic-based test criteria have been developed, including the active clause coverage (ACC) criteria and the modified condition/decision coverage (MCDC). The MCDC/ACC criteria is viewed as being expensive, which motivated us to evaluate the cost of applying these criteria using a basic proxy: the number of clauses. We looked at the frequency and percentage of predicates in 63 Java programs. Moreover, we also compared these Java programs with three programs in the safety-critical domain, in which logic-basic testing is often used. Although around 99% of the predicates within Java programs contain at most three clauses, there is a positive linear correlation between overall measures of size and the number of predicates that have more than three clauses. Furthermore, safety-critical C/C++ programs have more complex predicates than non-safety-critical programs. However, similar to the predicates in non-safety-critical programs, most predicates in safety-critical programs have up to three clauses. We conclude that non-safety-critical and safety-critical programs do not have many complex predicates. Thus, MCDC/ACC is only needed on a small fraction of the predicates.

1. Introduction

Logic predicates are ubiquitous in a myriad of software artifacts, including requirements, design models, test scripts, and source code. They are fundamental to the software's behavior, and define the possible flows of control through the program. The heart of any decision or branch is the logic predicate. If the predicate is wrong, the software behaves incorrectly. And the more subtle the mistake in the predicate is, the harder it is to find the mistake during testing and fix it during debugging.

Given this, it is not surprising that testers often use predicates to design tests. This is commonly called *logic-based testing* (Ammann and Offutt, 2008) and is applied during unit testing by designing tests from decisions in the code, and during integration and system testing by designing tests from decisions in the design or the requirements. Testers design tests to cause every predicate to become `true` and `false` (called *predicate coverage* – PC (Ammann and Offutt, 2008), decision coverage – DC (Myers et al., 2011), and branch coverage – BC), to cause every clause within every predicate to become `true` and `false` (*clause coverage* – CC and Condition Coverage – CC Myers et al., 2011), to cause every predicate to take on all of its possible truth values (*combinatorial coverage* – CoC), and to cause every clause to become `true` and `false` while the rest of the clauses have values to ensure the clause under test controls the value of the predicate (*modified condition, decision coverage* – MCDC and *active clause coverage* – ACC).

The US Federal Aviation Administration (FAA) has explicitly recognized the importance of logic-based testing by requiring that MCDC (Chilenski and Miller, 1994) be used to certify safety critical parts of the avionics software in commercial aircraft (RTCA-DO-178B, 1992).

Essentially, ACC is equivalent to MCDC. It turns out that MCDC's definition is ambiguous. The definition is not specific about the key notion of determination, the conditions under which a clause dictates the outcome of a predicate. This ambiguity has led to confusion in how to interpret MCDC. Resolving this ambiguity leads to three different interpretations of ACC: General ACC (GACC), Correlated ACC (CACC), and Restricted ACC (RACC)

* Corresponding author. Tel.: +55 1633738628.
E-mail addresses: vinicius.durelli@gmail.com, durelli@icmc.usp.br (V.H.S. Durelli), offutt@gmu.edu (J. Offutt), nli1@gmu.edu (N. Li), delamaro@icmc.usp.br (M.E. Delamaro), jguo_scce@home.swjtu.edu.cn (J. Guo), shizengshu@126.com (Z. Shi), axg1967@sina.com (X. Ai).

http://dx.doi.org/10.1016/j.jss.2015.12.022
0164-1212/© 2015 Elsevier Inc. All rights reserved.

Ammann et al. (2003). So, in this study, when we refer to ACC we are referring to all different interpretations of the criterion: GACC, CACC, and RACC. Both PC and CC require just two tests per predicate. ACC criteria and MCDC are more complicated to compute, making it harder to create quality test design tools and more time consuming to design tests by hand. ACC and MCDC also create up to $2n$ tests per predicate with n independent clauses, making it more expensive in terms of number of tests. But ACC and MCDC tests become more effective at finding faults as the number of clauses in the predicate increases.

If a predicate has a single clause, ACC, MCDC, and CC collapse to simply PC. If a predicate has two or three clauses, CoC is simpler to compute and not significantly more expensive (needing 2^n tests per predicate). However, as n grows, CoC quickly becomes prohibitively expensive, thus ACC is needed.

At the same time, designers and programmers tend to prefer simpler predicates, probably because it is easier to get these predicates correct. In addition, object-oriented (OO) programs tend to have fewer predicates than procedural programs because OO programs encode much of the conditional behavior in polymorphic method calls. Another factor that might contribute to simpler predicates in OO programs is the common use of refactoring (Fowler et al., 1999) to simplify the conditional expressions. This leaves a crucial question: how many clauses do predicates in real programs ("in the wild") have? If large predicates never occur, then more complicated logic-based test criteria such as ACC are not necessary. If they are very common, then high-end logic-based test criteria may be essential to effective testing. By effective testing, we mean applying cost-effective techniques with reasonable high fault-finding effectiveness. Some testers have criticized logic-based test criteria on the basis of its cost. However, the cost is low for small predicates, and only goes up when predicates have many clauses. Therefore, the question of the cost of logic-based test criteria is central to deciding their applicability.

Although predicates are used in many software artifacts (including control-flow graphs, finite state machines, UML diagrams, requirements, and source), and tests are designed from all of these artifacts, this research focuses on predicates derived from program source. The FAA requires that tests be generated from the requirements, but evaluated in terms of their MCDC coverage on the source (RTCA-DO-178B, 1992). This paper is concerned with the latter step, using predicates in code to either design or evaluate tests.

Our goal is to evaluate the cost of applying logic-based criteria. The cost of using a logic-based test criterion can be evaluated in several ways: *(i)* the number of tests, *(ii)* the cost of evaluating whether a set of tests satisfies the criterion, and *(iii)* the cost of generating tests to satisfy a criterion. We examined the predicates from 63 non-safety-critical Java programs, and asked two questions:

- How many clauses are in real predicates?
- Is the number of clauses per predicate correlated with the size of programs; that is, are complex predicates more likely to appear in large programs?

The overarching motivation for this research is to provide a greater understanding of the predicates found in real-world programs. Although previous papers Chilenski (2001); Kernighan and Plauger (1981); Chilenski and Miller (1994); Booch (1987) have analyzed the complexity of predicates, this paper presents the results of a large scale empirical study in a completely new light: besides looking at the frequency, percentage, and density of predicates with varying sizes, we also investigated the relationship between overall measures of size (number of lines of code and source files) and the frequency of predicates. To our knowledge, this is the largest and most rigorous analysis to date on predicates complexity. Previous studies have been much smaller, and are based on old programs in languages that are not currently used. This research updates our knowledge in the area by providing a significantly larger sample of evidence that corroborates previous research on predicates complexity. In addition, this paper compares non-safety-critical software with safety-critical software from the high-speed railway signaling domain.

The primary contribution of this research is a post-hoc empirical analysis of programs ranging from 423 to 629,114 lines of code investigating the characteristics of over 400,000 predicates. The results indicate that the vast majority of predicates have less than four clauses. We found a positive relationship between overall measures of size and the frequency of more complex predicates. Given that logic-based criteria are more often used in the safety critical domain, we also compared several Java programs to safety-critical C/C++ programs. According to our results, safety-critical systems have more predicates containing at least four clauses than non-safety-critical programs. Nevertheless, about 95% of the predicates in safety-critical systems have up to three clauses.

The remainder of this paper is organized as follows. Section 2 provides background on logic-based criteria, presents definitions for the terms and concepts in this paper, and discusses the cost of applying these test criteria. Section 3 describes the experimental design. Section 4 presents results, statistical analysis, compares Java and safety-critical systems, and discusses threats to validity. Section 5 compares our results with previous studies. Section 6 presents concluding remarks.

2. Background

This section introduces several definitions used in the rest of the paper. Most are taken from the textbook by Ammann and Offutt (Ammann and Offutt, 2008). The result of a *predicate* evaluation is a boolean value (`true` or `false`) (Ammann et al., 2003). Predicates may have boolean and non-boolean variables. Within predicates, relational operators ($>$, $<$, $\geq$, $\leq$, $=$, and $\neq$) are used to compare values and logical operators ($\wedge$, $\vee$, $\oplus$, $\rightarrow$, and $\leftrightarrow$) define the internal structure. An example predicate is the logical expression $(r \geq s) \wedge t$, where r and s are non-booleans and t is a boolean.

Predicates are comprised of one or more boolean-valued *clauses*, which are connected by logical operators (Ammann et al., 2003; Ammann and Offutt, 2008). For example, the predicate $(r \geq s) \wedge t$ contains two clauses, namely, a relational expression $(r \geq s)$ and a boolean variable t.

Predicates and clauses are used in several logic expression coverage criteria. The simplest is PC. Let P be a collection of predicates with clauses C. For each predicate $p \in P$, C_p is the set of clauses in p. PC can be defined as follows (Ammann and Offutt, 2008):

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

For the example $(r \geq s) \wedge t$, two tests that satisfy PC are $\{r = 6, s = 4, t = \texttt{true}\}$ and $\{r = 5, s = 10, t = \texttt{true}\}$. PC requires exactly two tests per predicate. A drawback is that it does not fully exercise individual clauses. For example, the clause t evaluates to `true` in both of these tests.

From a pragmatic testing standpoint, it is advantageous to evaluate the overall effect of each clause on a predicate. The simplest way is to evaluate predicates with all possible truth values. However, this creates 2^n tests, which gets prohibitively expensive very quickly. This thinking has led to several logic-based criteria to exercise clauses with a reasonable number of tests. These criteria draw on the idea of making clauses "active."