Information and Software Technology 55 (2013) 2125–2139

Contents lists available at ScienceDirect

Information and Software Technology

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

Estimating software testing complexity

Javier Ferrer *, Francisco Chicano, Enrique Alba

Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, E.T.S. Ingenieria Informatica, Campus de Teatinos, 29071 Málaga, Spain

ARTICLE INFO

Article history:
Received 17 December 2012
Received in revised form 19 July 2013
Accepted 20 July 2013
Available online 30 July 2013

Keywords:
Evolutionary testing
Complexity
Branch coverage
Search based software engineering
Evolutionary algorithms
Testability

ABSTRACT

Context: Complexity measures provide us some information about software artifacts. A measure of the difficulty of testing a piece of code could be very useful to take control about the test phase.
Objective: The aim in this paper is the definition of a new measure of the difficulty for a computer to generate test cases, we call it Branch Coverage Expectation (BCE). We also analyze the most common complexity measures and the most important features of a program. With this analysis we are trying to discover whether there exists a relationship between them and the code coverage of an automatically generated test suite.
Method: The definition of this measure is based on a Markov model of the program. This model is used not only to compute the BCE, but also to provide an estimation of the number of test cases needed to reach a given coverage level in the program. In order to check our proposal, we perform a theoretical validation and we carry out an empirical validation study using 2600 test programs.
Results: The results show that the previously existing measures are not so useful to estimate the difficulty of testing a program, because they are not highly correlated with the code coverage. Our proposed measure is much more correlated with the code coverage than the existing complexity measures.
Conclusion: The high correlation of our measure with the code coverage suggests that the BCE measure is a very promising way of measuring the difficulty to automatically test a program. Our proposed measure is useful for predicting the behavior of an automatic test case generator.

1. Introduction

Since the birth of Software Industry, there has been a high interest in measuring the effort in terms of time and cost required by a task. Nowadays, software applications are essential for Industry, thus software developers need to measure all sort of elements. Tom DeMarco stated [10]: "You can not control what you can not measure. Measurement is the prerequisite to management control". The importance of metrics have also been highlighted by the famous physicist Lord Kelvin [33]: "When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the state of science". For these reasons, in this work we focus on complexity measures, which quantify the effort required to complete any kind of task.

First, it is needed to define what program complexity means. Basili [5] defines complexity as a measure of the resources used by a system while interacting with a piece of software to perform a given task. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation described by the program. If the interacting system is a programmer then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing or modifying the software. There exist metrics introduced as all-purpose measures of software complexity, however these measures seem to be ineffective in order to measure the testing complexity [16]. The absence of a metric to properly measure the difficulty to test a piece of code encourage us to characterize the testing complexity.

Analyzing the testing complexity, it can be seen as the difficulty for a computer to create a test suite for finding errors in the developed code. Finding errors in early stages of the development is an important task that saves costs of the project. A detailed survey in the United States quantifies the high economic impacts of an inadequate software testing infrastructure [32]. Besides that, it is estimated that half the time spent on the software project development and more than half its cost, is devoted to testing the product [27]. To this end, in recent years researchers have attempted to predict fault-prone software modules using complexity metrics [36]. In addition, the overall experimental results show that complexity metrics are able to predict fault-prone source code [37].

In the last few years, there has been a renewed deal of interest in defining appropriate ways to measure the complexity of

* Corresponding author. Tel.: +34 952133303.
E-mail addresses: ferrer@lcc.uma.es (J. Ferrer), chicano@lcc.uma.es (F. Chicano), eat@lcc.uma.es (E. Alba).

0950-5849/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2013.07.007

software [15,26]. In most previous works they defined the testing complexity as the number of test cases required [35,22]. Some works try to compute the lower bound [7] of the test cases required, and other works try to provide better understanding on the testing criterion used to generate those test cases [23]. However, they do not focus on the effort to generate these test cases. In a recent work, Nogueira focuses on the correlation between the complexity of the software under test and the complexity of the test cases [28], but the work did not propose any estimation measure.

We propose in this work a new complexity measure with the aim of helping the tester to find errors in the code. This measure will predict in a better way the behavior of an automatic test data generator depending on the program under test. This original complexity measure, called "Branch Coverage Expectation", is the main contribution of this paper. The definition of the new measure lies on a Markov model that represents the program. Based on the model of a program, we can also provide an estimation of the number of random test cases that must be generated to obtain a concrete coverage. From these estimations, we can create a theoretical prediction of the evolution of the coverage depending on the number of generated test cases. This second contribution will help the testers to obtain some knowledge about the possible evolution of the testing phase.

The validation of the proposed measure is also addressed in this work. For the theoretical validation of the Branch Coverage Expectation we have used the validation framework proposed by Kitchenham et al. [19]. For the experimental validation we have used Evolutionary and Random Testing techniques, which are the most popular search algorithms for automatically generating test cases [1,2,12,21], to compare our estimation with the real value obtained by several test data generators.

Finally, we also analyze software complexity measures at program level and we discuss a number of issues associated with these known measures. In addition, we have performed an experimental study of correlations with the aim of highlighting the existing relationships among some static measures. We are especially interested in the existing relationships between the static measures and the branch coverage. In this experimental study we have used two large groups of automatically generated programs to serve as a benchmark.

The rest of the paper is organized as follows. In the next section we present the measures that we later use in our experimental study. In Section 3 we explain the Markov model on which two of our main contributions in this paper are based: the definition of the BCE measure and the estimation of the number of test cases required to obtain a particular branch coverage. In Section 4 we explain the details of the automatic test data generator and the benchmark of programs that we use in the experimental section. Later, Section 5 describes the experimental study performed. Towards the end of the article, we describe the threats to the validity of our experimental evaluation in Section 6. Finally, Section 7 outlines some conclusions and future work.

2. Static measures

Quantitative models are frequently used in different engineering disciplines for predicting situations, due dates, required cost, and so on. These quantitative models are based on some kind of measure made on project data or items. Software Engineering is not an exception. A lot of measures are defined in Software Engineering in order to predict software quality [30], task effort [8], etc. We are interested here in measures made on source code pieces. We distinguish two kinds of measures: *dynamic*, which require the execution of the program, and *static*, which do not.

Some time ago, project managers began to worry about concepts like productivity and quality, then the lines of code (LOC) metric was proposed. Nowadays, the LOC metric is still the primary quantitative measure in use. An examination of the main metrics reveals that most of them confuse the complexity of a program with its size. The underlying idea of these measures are that a program will be much more difficult to work with than a second one if, for example, it is twice the size, has twice as many control paths leading through it, or contains twice as many logical decisions. Unfortunately, these various ways in which a program may increase in complexity tend to move in unison, making it difficult to identify the multiple dimensions of complexity.

In this section we present the measures used in this study. In a first group we select the main measures that we found in the literature:

- Lines of Code (*LOC*)
- Source Lines of Code (*SLOC*)
- Lines of Code Equivalent (*LOCE*)
- Total Number of Disjunctions (*TNDj*)
- Total Number of Conjunctions (*TNCj*)
- Total Number of Equalities (*TNE*)
- Total Number of Inequalities (*TNI*)
- Total Number of Decisions (*TND*)
- Number of Atomic Conditions per Decision (*CpD*)
- Nesting Degree (*N*)
- Halstead's Complexity (*HD*)
- McCabe's Cyclomatic Complexity (*MC*)

Let's have a look at the measures that are directly based on source lines of code (in C-based languages). The *LOC* measure is a count of the number of semicolons in a method, excluding those within comments and string literals. The *SLOC* measure counts the source lines that contain executable statements, declarations, and/or compiler directives. However, comments, and blank lines are excluded. The *LOCE* measure [31] is based on the idea of weighing each source line of code depending on how nested it is. The previous three measures based on the lines of code have several disadvantages:

- Depend on the print length
- Depend of the programmer's style for writing source code
- Depend on how many statements does one put in one line

We have analyzed several measures as the total number of disjunctions (OR operator) and conjunctions (AND operator) that appear in the source code, these operators join atomic conditions. The number of (in) equalities is the number of times that the operator (!=) = = is found in atomic conditions of a program. The total number of decisions and the number of atomic conditions per decision do not require any comment. The nesting degree is the maximum number of control flow statements that are nested one inside another. In the following paragraphs we describe the McCabe's cyclomatic complexity and the Halstead complexity measures in detail.

Halstead complexity measures are software metrics [14] introduced by Maurice Howard Halstead in 1977. Halstead's Metrics are based on arguments derived from common sense, information theory and psychology. The metrics are based on four easily measurable properties of the program, which are:

- $n1$ = the number of distinct operators
- $n2$ = the number of distinct operands
- $N1$ = the total number of operators
- $N2$ = the total number of operands