A Simple Complexity Measurement for Software Verification and Software Testing

- Discussion Paper -

Zheng Cheng*, Rosemary Monahan, and James F. Power

Computer Science Department
National University of Ireland Maynooth
Co. Kildare, Ireland
{zcheng,rosemary,jpower}@cs.nuim.ie

Abstract. In this paper, we used a simple metric (i.e. Lines of Code) to measure the complexity involved in software verification and software testing. The goal is then, to argue for software verification over software testing, and motivate a discussion of how to reduce the complexity involved in software verification. We propose to reduce this complexity by translating the software to a simple intermediate representation which can be verified using an efficient verifier, such as Boogie [2].

Keywords: Intermediate Verification Language, Software Testing, Software Verification, Metrics

1 Introduction

Software testing cannot guarantee that a tested program will be free from errors, whereas software verification can [1]. However, full verification of entire systems is often impossible because of resource limitations and complexity. Therefore, software testing is still the most common technique for ensuring the reliability of software systems [11].

In this paper, we discuss a possible metric for comparing the complexity of software verification and software testing. The goal is to motivate a discussion of how to reduce the complexity involved in software verification, thereby making verification more applicable for industrial usage.

Using different metrics to measure the complexity of software programs is an active research area, see [4] for an overview. We propose using the Lines of Code (LoC) metric to measure the complexity involved in software verification and software testing. The result of LoC-based complexity measurement shows that if using an efficient program verifier, software verification can be no harder than software testing, while improving our confidence in the correctness of our software. We argue that the Boogie verifier is a suitable verifier and suggest how to correctly translate software to a suitable intermediate representation for input to this verifier.

* Funded by John & Pat Hume Scholarship and Doctoral Teaching Scholarship from the Computer Science Department of NUI Maynooth.

Outline A case study is presented in Section 2. The discussion topic of this paper and our view to it are both suggested in Section 3. Finally, the conclusion is made in Section 4.

2 Case Study

As a case study, we demonstrate a two-way sorting algorithm, taken from the software verification competition of VSTTE 2012 [10], and see how many LoC[1] are written to complete its verification task and its testing task[2]. By "complete", we mean that all the pre-defined criteria for each task have been checked thoroughly.

2.1 Criteria Setting

First, we set one criterion for software testing, i.e., functional behaviour (ensuring an array of booleans is sorted in the given order). Then, we manually generate code to meet the criterion. This code is usually referred to as "test cases" and can be executed by a test runner (in this case, the code targets NUnit [9]). We count the LoC of all test cases as the complexity measurement for software testing.

Next, we set three criteria for software verification, i.e. functional behaviour, termination and safety (e.g. ensure no invalid array access). Then, we manually generate annotations (e.g. preconditions, postconditions and loop invariants) to meet the criteria. These annotations can be interpreted by an automated program verifier (in our case we use Dafny [6]). We count the LoC of annotations as the complexity measurement result for software verification. Regarding the conjunction(s) in a proof obligation, the LoC would count the conjunction symbols (e.g. ampersand symbol) plus 1. For example, in Dafny, a postcondition expressed as "requires A & B;", which would count as two LoC. We have to admit that the current methodology for counting LoC of software verification is informal, and requires further research to make it formalized.

2.2 Results

The LoC measurement results for software verification and software testing are listed in Table 1.

Generally, it is infeasible to test the absence of an event [7]. Thus, the termination and safety criteria are more appropriate for verification than testing. For example, if a program executes and does not stop, all that we know is that the program has not halted yet and no conclusion can be derived from such a circumstance. Whereas in program verification, proof of program termination

[1] The LoC is counted by logical line of code, i.e. a statement followed by a domain-specific termination symbol (e.g. semicolon) will count as one logical line of code.

[2] The full description of this case study can be found at:
http://www.cs.nuim.ie/∼zcheng/COMPARE2012/case.html

Question under Study: Two-way Sort Algorithm				
	Functional Behaviour	Termination	Safety	Total
Software Testing	16	N/A	N/A	16
Software Verification	13	0	2	15

Table 1: LoC measurement result for the Two-way Sort Algorithm

can ensure a program will always halt. For example, the Dafny program verifier uses the keyword ***decreases*** to express the variant functions which are used to proof termination. It is also capable of automatically guessing simple variant functions.

Regarding the functional behaviour criterion, we can see the LoC for software testing is greater than for software verification. Moreover, software testing by nature cannot guarantee all the circumstances are tested. Therefore, in order to get more confidence about a program under test, new code (i.e. test cases) is needed. In contrast, the LoC for functional behaviour checking in software verification is a fixed number (i.e. no extra annotations are needed once a program is verified).

One approach for reducing the LoC involved in software verification is using an intermediate verification language such as the Boogie language [2]. For example, the Dafny program verifier translates its program and specifications into the Boogie language, which allows the Dafny program verifier to use the Boogie verifier as its back-end. The Boogie verifier features abstract interpretation for inference of properties such as loop invariants. Moreover, mathematical theories (e.g., set theory and tree theory) are encoded in the Boogie language in advance, which allows Dafny program verifier writing concise model-based specifications. All these features of intermediate verification language can reduce the quantity of annotations that must be discharged in the verification process. Related work shows that program verifiers powered by the Boogie verifier are excellent in accuracy and efficiency [8].

3 How to Reduce the Complexity of Software Verification

We think using a suitable program verifier can lower the complexity of software verification. In [5], we proposed a reliable generic translation framework for the Boogie language (shown in Figure 1), allowing convenient access to the Boogie verifier. The modelling and metamodelling approach [3] provides the foundation of the framework. An intermediate representation, i.e. the Boogie Extension Metamodel, is introduced to bridge the translation from different source languages to the Boogie language, thereby reducing the translation complexity. By the assistance of proposed framework, it is expected that software verification would be accessible for software developers even more.

We also believe that there are many potential solutions to reduce the complexity of software verification, and further discussion on this topic is warranted.

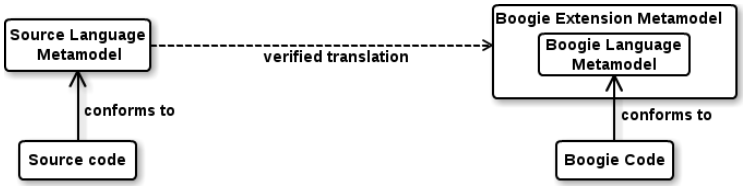

Fig. 1: Overview of Our Proposed Generic Translation Framework

4 Conclusion

In this paper, we used a simple metric (i.e. LoC) to measure the complexity involved in software verification and software testing. The result motivates the use of software verification over software testing, and shows that an efficient program verifier can greatly reduce the verification complexity. How to reduce the complexity of software verification is still an open question that deserves further discussion. In our opinion, the Boogie verifier is a suitable verifier for efficient software verification. To interact with the Boogie verifier, a Boogie program is required as the intermediate representation of the source program to be verified. Our proposed translation framework, based on metamodelling, provides the ideal platform for a reliable translation from a source program to a Boogie program.

References

1. Barnes, J.: High integrity software: The Spark approach to safety and security. Addison-Wesley (2003)
2. Barnett, M., Chang, B.Y.E., Deline, R., Jacob, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. Lecture Notes in Computer Science Vol. 4111 (2006)
3. Bézivin, J.: In search of a basic principle for Model-Driven Engineering. UPGRADE Vol. 5, No. 2 (2004)
4. Cardoso, J., Mendling, J., Neumann, G., Reijers, H.A.: A discourse on complexity of process models (survey paper). Lecture Notes in Computer Science Vol. 4103 (2006)
5. Cheng, Z.: A proposal for a generic translation framework for Boogie language. In: European Conference on Object-Oriented Programming, PhD Workshop (2012)
6. Dafny: http://research.microsoft.com/en-us/projects/dafny/
7. Dijkstra, E.W.: Notes On Structured Programming. Academic Press Ltd. (1972)
8. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: Experience report. In: Symposium on Formal Methods, Limerick, Ireland (2011)
9. NUnit: http://www.nunit.org/
10. VSTTE.2012.Competition: https://sites.google.com/site/vstte2012/compet/
11. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Computing Surveys Vol. 41, No.4 (2009)