Improving Test Coverage of Formal Verification Systems via Beam Search

Mahmoud Bokhari
Computer Science School
University of Adelaide
Adelaide, Australia

Computer Science Department
Taibah University
Medina, Kingdom of Saudi Arabia

Markus Wagner
Optimisation and Logistics,
University of Adelaide
Adelaide, Australia

ABSTRACT

The correctness of program verification systems is of great importance, since they are used to formally prove that safety- and security-critical programs follow their specification. Within these verification systems, the background axiomatization captures the semantics of the target program language—errors here can lead to incorrect formal proofs, which in turn can have devastating consequences.

Testing the axiomatization thoroughly is one approach to increase the trust in its correctness, however, the manual creation of test cases is a very time-consuming process for verification engineers. We present a beam search approach for creating test cases through test case modifications.

Categories and Subject Descriptors

G.1.6 [**Optimization**]: Miscellaneous; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic Methods*

1. INTRODUCTION

Broadly speaking, the purpose of software verification is to prove or disprove the correctness of software using formal or informal methods [8]. Examples of formal methods that use mathematical approaches include model checking and deductive verification [1]. Due to the error-prone nature of manually applying such methods, the need for auto-active verification software systems arises.

Deductive verification systems following the auto-active verification paradigm must understand their target programs, which can be accomplished by using the so-called axiomatization base. Consisting of a large set of axioms (typically 100s of axioms), it carries the semantics of the programming language used for writing the target programs.
Motivation. To confirm the correctness of software verifiers, it is necessary to test both parts: the axiomatization base and the implementation [6]. Only testing the latter is not sufficient, even if high code coverage is achieved. For instance, the authors of [3] reported that there is a relatively high amount of "core code" executed by the test suite, however, a small number of "core axioms" were used during the test. Some logical bugs remain uncovered within the axiomatization unless it is fully exercised. In addition, two bugs in the axiomatization were found during the coverage maximization research in [3].

GECCO '15 July 11-15, 2015, Madrid, Spain

ACM ISBN 978-1-4503-3488-4/15/07.
DOI: http://dx.doi.org/10.1145/2739482.2764670

Axiom	Replacement Set	Successful Times	Total Successful Times	Unsuccessful Times
ax1		0	0	10
ax2	{ax4}	10	10	7
ax3	{ax5}	6	8	0
	{ax6, ax7}	2		3

Table 1: Guidance Table example

In order to maximize the axiomatization coverage, a large number of test cases are required. In our situation, test cases consist of programs and their formal specifications. Creating such test cases manually from scratch is a difficult and time consuming task even for experienced verification engineers, due to their inherent complexity.
Topic. We present a heuristic approach to maximize the number of axioms used in successful proof procedures [2]. Our approach utilizes the beam search algorithm [5], which uses some historical data to explore the search space. We use such an algorithm to address two main key issues: having a large number of axioms as well as the time consuming verification process (sometimes minutes).

The author of [9] presents a collection of uniformed approaches that explore the search space in breadth-first and depth-first fashions to increase the axiomatization coverage. In contrast to this, our approach is informed by previous runs. We do this in order to achieve two goals: minimizing the likelihood of creating infeasible solutions, and maximizing the likelihood of covering previously uncovered axioms.

2. ALGORITHM

Guidance Table. To effectively explore the search space, our approach uses a guidance table GT to inform and guide the search process towards promising nodes. The GT keeps track of the following: each used axiom in every minimal set, its successful replacement sets, the total number of successful replacements for each set, the total number of all successful replacements, and the total number of unsuccessful attempts for replacing the axiom. All data is recorded for each axiom. Table 1 presents an example of the guidance table: it shows that the two sets $\{ax5\}$ and $\{ax7, ax6\}$ successfully replaced *ax3* eight times in total. While the former replaced it six times, the latter was only used twice as a replacement for it.

The GT is also used to discover equivalences between ax-

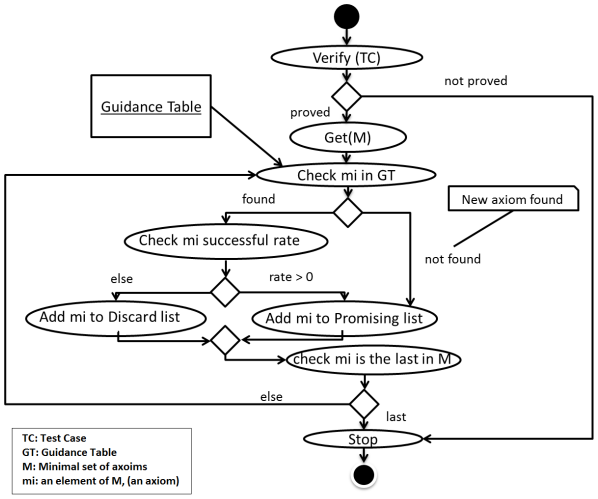

Figure 1: Activity Diagram: Beam Search.

ioms.[1] In other words, it records the equivalent sets of axioms for each individual axiom or axiom set. For example, Table 1 shows that *ax3* has two equivalent sets of axioms which are {*ax5*} and {*ax6*, *ax7*}. It is important to note that these two sets are only equivalent to that axiom in eight cases in total. However, since they are not entirely equal to *ax3*, the second set could not replace *ax3* in three instances.

Beam Search. Diagram 1 illustrates our informed beam search. As can be seen, it tries to verify the test case TC. If it cannot find a proof it stops otherwise it continues to the next step, which is receiving the minimal set of axioms used in the proof M. Then for each axiom mi within M, it checks whether it is recorded in the GT. In case the mi is not found, it is added to the promising list. This is because the GT contains all previously discovered axioms and therefore this mi is recently covered and has to be investigated first.

On the other hand, if the mi is found in the GT, it is added to the promising list only if it has been successfully replaced, i.e. the successful rate is greater than zero. However, if it has not been replaced at all we add it to the discard list.

3. APPROACH ANALYSIS

Now let us look to how our approach explores the search space. Figure 2 illustrates a small part of one test case's search space. At the beginning the java card software verifier KeY system starts the proof procedure using 1520 axioms[1]. It successfully finds a proof for the given test case using 31 axioms, which represents the first minimal set M.

Using the GT, our approach explores the promising nodes to be dropped from the axoimatization base. This operation forces the KeY to find an alternative proof for the same test case, but without the dropped axioms; as a result, the coverage increases. We refer to this method by white and black listing, where the white list includes what the verifier can use, whereas the black list contains the forbidden axioms.

Initially, the approach identifies four axioms to be dropped *a1, a2, a15* and *a31*. After excluding each axiom, the verifier successfully manage to prove the same test case. As a result, four *Ms* are found, however, in this case, two of them are

[1]"Equivalence" is not strictly logical here, but regarding the tool's capability to find a proof in a different way.

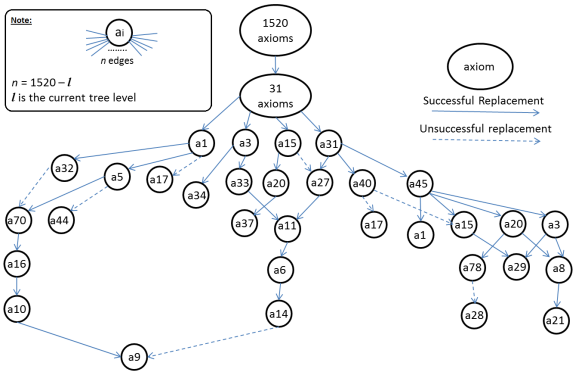

Figure 2: Search Space Example: Exploring the search space by the beam search approach.

equal to the first M. In other words, the other two *Ms* contains new axioms that are not covered in the first M.

In the subsequent stage, the approach tries to identify new promising nodes to drop together with the previous ones, i.e it drops pairs of axioms. It chooses 9 axioms in total, nevertheless, only two pairs are unsuccessfully replaced {*a1, a17*} and {*a15, a27*}. Although the last set could not be replaced, it was possible to replace it after adding one axiom *a11* to it. This is because the replacement of *a15, a11* renders *a27* redundant for the proof.

It is worth mentioning that our approach identifies the first four nodes without trying to drop the 1520 axioms, which significantly reduces the time complexity. Furthermore, in total it tries only 38 replacements where 79% of them are successful.

In-depth studies will follow in the very near future.

References

[1] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, Vol. 4334 of *LNCS*. Springer, 2007.

[2] B. Beckert, T. Bormer, and M. Wagner. Heuristically creating test cases for program verification systems. In *Metaheuristics Int. Conference (MIC)*, Singapore, 2013.

[3] B. Beckert, M. Wagner, and T. Bormer. A metric for testing program verification systems. In *Tests and Proofs (TAP)*, Vol. 7942 of *LNCS*, pp. 56–75, 2013.

[4] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer, 2010.

[5] C. Blum and M. J. Blesa. Probabilistic beam search for the longest common subsequence problem. In *Stochastic Local Search Algorithms (SLS)*, pp. 150–161, Berlin, Heidelberg, 2007. Springer.

[6] T. Bormer and M. Wagner. Towards testing a verifying compiler. In *Int. Conference on Formal Verification of Object-Oriented Software (FoVeOOS). Pre-Proceedings*, pp. 98–112. Karlsruhe Institute of Technology, 2010.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[8] G. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. ITPro collection. Wiley, 2011.

[9] M. Wagner. Maximising axiomatization coverage and minimizing regression testing time. In *IEEE Congress on Evolutionary Computation (CEC)*, pp. 2885–2892, 2014.