World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:8, No:12, 2014

A New Approach for Assertions Processing during Assertion-Based Software Testing

Ali M. Alakeel

***Abstract*—**Assertion-Based software testing has been shown to be a promising tool for generating test cases that reveal program faults. Because the number of assertions may be very large for industry-size programs, one of the main concerns to the applicability of assertion-based testing is the amount of search time required to explore a large number of assertions. This paper presents a new approach for assertions exploration during the process of Assertion-Based software testing. Our initial exterminations with the proposed approach show that the performance of Assertion-Based testing may be improved, therefore, making this approach more efficient when applied on programs with large number of assertions.

***Keywords*—**Software testing, assertion-based testing, program assertions.

I. Introduction

RESEARCH has shown that software testing is a very labor intensive and tedious task [12]. There are two main approaches to software testing: Black-box and White-box. Test generators that support black-box testing create test cases by using a set of rules and procedures; the most popular methods include equivalence class partitioning, boundary value analysis, cause-effect graphing. White-box testing is supported by coverage analyzers that assess the coverage of test cases with respect to executed statements, branches, paths, etc. There are different types of automated test data generators for white-box testing. Random test data generators select random inputs for the test data from some distribution, e.g., [10]. Path-oriented test data generators select a program path(s) to the selected statement and then generate input data to traverse that path, e.g., [1], [3], [16], [19], [20]. Goal-oriented test data generators select inputs to execute the selected goal (i.e. statement) irrespective of the path taken, e.g., [4], [6], [21]). Intelligent test data generators employ genetic and evolutionary algorithms in the process of generating test data, e.g., [2], [9], [15], [18], [22]. Assertions have been recognized as a powerful tool for automatic run-time detection of software errors during debugging, testing, and maintenance [7], [8], [14], [17], [23]. An assertion specifies a constraint that applies to some state of a computation. When an assertion evaluates to *false* during program execution, there exists an incorrect state in the program. Moreover, assertions have proved to be very effective in testing and debugging cycle [11]. For example, during black-box and white-box testing assertions are evaluated for each program execution [6]. Information about assertion violations is used to localize and fix bugs [11]-[24], and can increase program's testability [13], [14].

Utilizing assertions for the purpose of test data generation was proposed in [6]. In that research, an automated test data generation method based on the violation of assertions was presented. The main objective of this method is to find an input on which an assertion is violated. If such an input is found then there is a fault in the program. This type of assertion-based testing is a promising approach as most programming languages nowadays support automatic assertions generation. Examples of automatically generated assertions are boundary checks, division by zero, null pointers, variable overflow/underflow, etc.

Because the number of assertions may be very large for industry-size programs, one of the main concerns to the applicability of assertion-based testing is the amount of search time required to explore a large number of assertions. This paper presents a new approach for assertion exploration during the process of Assertion-Based software testing. Our initial exterminations with the proposed approach show that the performance of Assertion-Based testing may be improved, therefore, making this approach more efficient when applied on programs with large number of assertions.

The rest of this paper is organized as follows. Section II provides an overview of Assertion-Based software testing. Section III presents our proposed approach for processing large number of assertions during Assertion-Based testing. In Section IV, we discuss our conclusions and future research.

II. Assertion-Based Software Testing

The goal of Assertion-Based software testing [6] is to identify program input on which an assertion(s) is violated. This method is a goal-oriented [4], [5], [21] and is based on the actual program execution. This method reduces the problem of test data generation to the problem of finding input data to execute a *target* program's statement s. In this method, each assertion is eventually represented by a set of program's statements (nodes). The execution of any of these nodes causes the violation of this assertion. In order to generate input data to execute a target statement s (node), this method uses the chaining approach [21]. Given a target program statement s, the chaining approach starts by executing the program for an arbitrary input. When the target statement s is not executed on this input, a fitness function [4], [5], [21] is associated with this statement and function minimization

Ali M. Alakeel is an associate professor of computer science in the Faculty of Computers and Information Technology, University of Tabuk, P.O.Box 741, Tabuk 71491, Saudi Arabia (e-mail: alakeel@ut.edu.sa).

search algorithms are used to find automatically input to execute s. If the search process can't find program input to execute s, this method identifies program's statements that have to be executed prior to reaching the target statement s. In this way this approach builds a chain of goals that have to be satisfied before the execution to the target statement s. More details of the chaining approach can be found in [21].

```
program sample;
var
n: integer;
a: array[1..10] of integer;
i,max,min: integer;
begin
1       input(n,a);
2       max:=a[1];
3       min:=a[1];
4       i:=2;
5       while i ≤ n do begin
6,7             if min > a[i] then min:=a[i];
8               i:=i+1;
        {Assertion A1 as a Boolean formula}
                (*@  (i ≥ 1) and (i ≤ 10) @*)
9,10            if max < a[i] then max:=a[i];
        end;
        {Assertion A2 as executable code}
         (*@ assertion:
         var
         j: integer;
         begin
            assert:=true;
            j:=1;
            while j ≤ n do begin
               if max < a[j] then assert:=false;
               j:=j+1;
            end;
         end;
         @*)

11      writeln(min,max);
end.
```

Fig. 1 Sample program with assertions

As presented in [6], two types of assertions are dealt with: Boolean-formula and Executable-code assertions. As demonstrated using Pascal programs, each assertions is written inside Pascal comment regions using the extended comment indicators: (*@ assertion @*) in order to be replaced by an actual code and inserted into the program during a preprocessing stage of the program under test. Fig. 1 shows a sample Java method with assertions. This simple method computes the maximum and minimum element of a set of integers. An assertion may be described as a Boolean formula built from the logical expressions and from (**and**, **or**, **not)** operators. In our implementation we use Pascal language notation to describe logical expressions. There are two types of logical expressions: Boolean expression and relational expression. A Boolean expression involves Boolean variables and has the following form: A_1 *op* A_2, where A_1 and A_2 are Boolean variables or *true*/*false* constant, and *op* is one of {=, ≠}. On the other hand, relational expression has the following form: A_1 *op* A_2, where A_1 and A_2 are arithmetic expressions, and *op* is one of {<, ≤, >, ≥, =, ≠}. For example, (x < y) is a relational expression, and (f = false) is a Boolean expression.

The following is a sample assertion:

A: (*@ (x < y) **and** (f = false) @*).

The preprocessor in our implementation translates assertion *A* into the following code:

```
if not ((x < y) and (f = false)) then
Report_Violation;
```

where, Report_Violation, is a special procedure which is called to report assertion's violation.

III. The Proposed New Approach for Assertion Processing

As presented in [6], each program assertion, A, may be replaced by a block of conditional statements as in Fig. 2.

```
IF  c11 THEN
   IF c12 THEN
      …
      IF c1r THEN n1;
IF c21 THEN
   IF c22 THEN
      …
      IF c2r THEN n2;
…
IF cz1 THEN
   IF cz2 THEN
      …
      IF czr THEN nq;
```

Fig. 2 The Corresponding code generated for an example assertion

For our this presentation, let A = {A_1, A_2, …, A_n} be a set of assertions found in a program P. For each assertion A ∈ A, a set of nodes N(A) = {n_1, n_2, …, n_q} where q ≥ 1, is identified during a preprocessing stage of the program under test, where the execution of any node n_k ∈ N(A), 1≤k≤q, corresponds to the violation of assertion A. In other words, an assertion A is violated if and only if there exists a program input data **x** for which at least one node n_k ∈ N(A) is executed. Furthermore, with each node n_k ∈ N(A) we associate a sequence of nested-if conditions C(n_k) =< c_1, c_2, …, c_r> where r ≥ 1, which leads to node n_k. For node n_k to be executed, every condition c_l ∈ C(n_k), 1≤*l* ≤r, has to be satisfied.

For example, Fig. 3 shows code statements generated to represent the following assertion *A*:

(*@ ((x≥y) **or** (x=z)) **and** ((z≠99) **or** (Full=False)) **and** (z≠0) @*), where,
N(*A*) = { n_1, n_2, n_3 },
C(n_1) = < (x < y), (x ≠ z) >,
C(n_2) = < (z = 99), (Full = True) >, and
C(n_3) = < (z = 0) >.

In order for assertion *A* to be violated we have to find a program input **x** that will cause *at least* one of n_1, n_2, or n_3 to be executed.

```
IF (x < y) THEN
    IF (x ≠ z) THEN
n1 Report_Violation;
  IF (z = 99) THEN
    IF (Full = True) THEN
n2 Report_Violation;
  IF (z = 0) THEN
n3 Report Violation;
```

Fig. 3 Code generated for an example assertion *A*

During Assertion-Based testing [6], the exploration of assertions found in the program is conducted in a sequential manner one by one in an attempt to find an assertion's violation. The problem with this sequential approach is that valuable search time may be wasted while trying to violate some assertions blindly. This problem is aggravated when the number of assertions is very large as expected to be the case when testing industry-size programs. In order to deal with this problem and to make Assertion-Based software testing [6] more efficient and effective in the presence of large number of assertions, we propose to process assertions found in the program as follows.

The proposed approach analyzes results of previously processed assertions or nodes and then tries to employ this result while processing new assertions in the future. Depending on the result of the current exploration the proposed approach decides on how to move in the next step as follows. If a violation of the current assertion is reached then move to the next one sequentially. However, if a violation is not achieved in the allocated time, then this approach will perform an analysis process in order to decide which assertion to explore in the next step. This analysis process is based on finding data dependencies [21] among pairs of unprocessed assertions with the objective to project the results of assertion's violation in the next step. Specifically, this analysis has two main goals. The first goal is to explore the possibility of violating more than one assertion based on the same input data **x**. The second goal is to perform data-dependency analysis [21] among assertions to identify assertion nodes that have the potential to be executed and give them a higher priority during test data generation. To reach the first goal, program's execution is performed to the end every time the system succeeds in finding input data **x** to violate an assertion. This action is done in the hope that assertion nodes identical or related to the one which caused the violation of the currently explored assertion will also be executed based in the same input data. By doing so, this approach may be able to reduce the number of assertions to be explored which will consequently results in reducing the cost associated with assertion-based test data generation. Two nodes n_k and n_p are related if the conditional sequence of n_p is contained in the conditional sequence of n_k or vice versa.

In order to satisfy the second goal, i.e., to identify nodes with high potential to be executed, data dependency analysis is conducted after every program execution in order to identify which assertion nodes should be given priority to be explored first in the next execution. Because this analysis is conducted after each assertion' node n_k was executed, the objective of this step is three fold. First, given a previously executed node n_k, for every assertion H in the set R of yet to be explored assertions, identify every node $n_p \in N(H)$ for which the conditional sequence $C(n_p)$ is identical or a subsequence of the conditional sequence $C(n_k)$ of node n_k. Second, collect data-dependency analysis to check whether or not any of the variables used at $C(n_p)$ has been modified between node n_k and node n_p. Third, if the result of this analysis shows that all variables used at $C(n_p)$ were not modified between node n_k and node n_p, then node n_p is considered as a *candidate* to be executed first in the next iteration and is assigned a priority number to distinguish it from other nodes. Our priority system is very simple where a candidate node is simply moved to the head of the list of nodes to be explored.

Our initial experimentation with the proposed approach shows the proposed algorithm may be able to save valuable search time, hence making Assertion-Based software testing more efficient and applicable on large software with large number of assertions.

IV. Conclusion

In this paper, we have presented a new approach for assertions processing during the process of Assertion-Based software testing. The main goal of the proposed approach is to make Assertion-Based testing more efficient in the presence of large number of assertions that may exist in large programs. This goal is achieved by saving valuable searching resources during the process of assertions exploration. Our initial experimentations with the proposed approach show that this approach may succeed in reducing the amount of search time required to explore an assertion, therefore, making Assertion-Based software testing more effective and efficient when applied on larger programs. For our future research, we intend to perform a set of experiments in order to evaluate the performance of the proposed approach when applied on software with large number of assertions.

References

[1] C. Ramamoorthy, S. Ho, W. Chen, "On the Automated Generation of Program Test Data," IEEE Transactions on Software Engineering, vol. 2, No. 4, 1976, pp. 293-300.
[2] B. Jones, H. Sthamer, D. Eyres, "Automatic Structural Testing Using Genetic Algorithms," Software Eng. Journal, 11(5), 1996, pp.299-306.
[3] B. Korel, "Automated Test Data Generation," IEEE Transactions on Software Engineering, vol. 16, No. 8, 1990, pp. 870-879.
[4] B. Korel, "Dynamic Method for Software Test Data Generation," Journal of Software Testing, Verification, and Reliability, vol. 2, 1992, pp. 203-213.
[5] B. Korel, "TESTGEN – An Execution-Oriented Test Data Generation System," Technical Report TR-SE-95-01, Dept. of Computer Science, Illinois Institute of Technology, 1995.

[6] B. Korel, A. Al-Yami "Assertion-Oriented Automated Test Data Generation," Proc. 18th Intern. Conference on Software Eng., Berlin, Germany, 1996, pp. 701-80.
[7] B. Korel, , Q. Zhang, L. Tao, "Assertion-Based Validation of Modified Programs," Proc. 2009 2nd Intern. Conference on Software Testing, Verification and Validation, Denver, USA, 2009, pp. 426-435.
[8] Ali M. Alakeel, "Using Fuzzy Logic in Test Case Prioritization for Regression Testing Programs with Assertions," The Scientific World Journal, vol. 2014, Article ID 316014, 9 pages, 2014. doi:10.1155/2014/316014.
[9] C. Michael, G. Mcgraw, M. Schatz., "Generating Software Test Data by Evolution," IEEE Tran. on Software Engineering, 27(12), 2001, pp. 1085-1110.
[10] D. Bird, C. Munoz, "Automatic Generation of Random Self-Checking Test Cases," IBM Systems Journal, vol. 22, No. 3, 1982, pp. 229-245.
[11] D. Rosenblum, "Toward A Method of Programming With Assertions," Proceedings of the International Conference on Software Engineering, 1992, pp. 92-104.
[12] G. Myers, "The Art of Software Testing," John Wiley & Sons, New York, 1979.
[13] Ali M. Alakeel, "A Testability Transformation Approach for Programs with Assertions," Proceedings of the Sixth International Conference on Advances in System Testing and Validation Lifecycle, Nice, France, pp. 9-13, October 2014.
[14] Ali M. Alakeel, "Intelligent Assertions Placement Scheme for String Search Algorithms," Proceedings of the Second International Conference on Intelligent Systems and Applications, Venice, Italy, pp. 122-128, April 2013.
[15] J. Wegener, A. Baresel, H. Sthamer, "Evolutionary Test Environment for Automatic Structural Testing," Information and Software Technology, 43, 2001, pp. 841-854.
[16] L. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, vol. 2, No. 3, 1976, pp. 215-222.
[17] Ali M. Alakeel, "Assertion-Based Software Testing Metrics Approach Based on Fuzzy Logic," Proceedings of the 22nd International Conference on Software Engineering and Data Engineering (SEDE–2013), Los Angeles, California, USA, pp. 9-12, September 2013.
[18] P. Mcminn, M. Holcombo, "The State Problem for Evolutionary Testing," Proc. Genetic and Evolutionary Computation Conference, 2003, pp. 2488-2498.
[19] R. Boyer, B. Elspas, K. Levitt, "SELECT - A Formal System for Testing and Debugging Programs By Symbolic Execution," SIGPLAN Notices, vol. 10, No. 6, 1975, pp. 234-245.
[20] R. DeMillo, A. Offutt, "Constraint-Based Automatic Test Data Generation," IEEE Transactions on Software Engineering, vol. 17, No. 9, 1991, pp. 900-910.
[21] R. Ferguson, B. Korel, "Chaining Approach for Automated Test Data Generation," ACM Tran. on Software Eng. and Methodology, (5)1, 1996, pp.63-68.
[22] R. Pargas, M. Harrold, R. Peck, "Test Data Generation Using Genetic Algorithms," Journal of Software Testing, Verification, and Reliability, 9, 1999, pp. 263-282.
[23] S. Yau, R. Cheung, "Design of Self-Checking Software," Proceedings of the International Conference on Reliable Software, 1975, pp. 450-457.
[24] N. Levenson, S. Cha, J Knight, T. Shimeall, "The Use of Self Checks and Voting in Software Error Detection: An empirical study," IEEE Trans. on Software Eng., 16(4), 1990, pp. 432-443.