# A Testability Transformation Approach for Programs with Assertions

Ali M. Alakeel
Department of Computer Science
University of Tabuk
Tabuk, Saudi Arabia
e-mail: alakeel@ut.edu.sa

*Abstract*— **Assertion-Based software testing has been shown to be effective in detecting program faults as compared to traditional black-box and white-box software testing methods; however in the presence of large numbers of assertions this approach may be very expensive. As reported in the literature, Assertion-Based software testing executes the whole program based on a given input data in order to find an assertion's violation. Executing the whole program for every assertion may be very costly especially for large programs with very larger number of assertions. The cost is related to search time required during the process of generating test input data to violate such large number of assertions. This paper introduces a testability transformation approach based on the analysis of control and data flow dependencies that affect the execution of every assertion in the program. It achieves this by eliminating program statements that do not lead the program flow control to the assertion under consideration. A small case study is presented, which demonstrates the value of the proposed approach.**

***Keywords-assertion-based software testing; testability transformation; software testing; data dependency analysis***

## I. INTRODUCTION

Software testing is the process of executing a program with the intent of detecting faults [1]. Software testing is a very labor intensive and tedious task. For this reason, many studies have been devoted to the automation software testing [2]-[7]. There are two main approaches to software testing: Black-box and White-box [1]. Test data generation is the process of finding program input data that satisfies a given criteria. 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 [1]. White-box testing is supported by coverage analyzers that assess the coverage of test cases with respect to executed statements, branches, paths, etc. Programmers usually start by testing software using black-box methods against a given specification. By their nature, black-box testing methods might not lead to the execution of all parts of the code. Therefore, this method may not uncover all faults in the program. To increase the possibility of uncovering program faults, white-box testing is then used to ensure that an acceptable coverage has been reached, e.g., branch coverage.

Assertion-based software testing [9]-[10] has been shown to be effective in detecting program faults as compared to traditional black-box and white-box software testing methods. The main objective of assertion-based testing is to find a program input on which an assertion is violated. If such an input is found then there is a fault in the program. Some programming languages support assertions by default, e.g., Java [21] and Perl [22]. For languages without built-in support, assertions can be added in the form of annotated statements. In [9], assertions are represented as commented statements that are pre-processed and converted into Pascal code before compilation. Many types of assertions can be easily generated automatically such as boundary checks, division by zero, null pointers, variable overflow/underflow, etc. Therefore, programmers may be encouraged to write more assertions in their programs in order to enhance their confidence in their programs.

As reported by Korel and Al-Yami [9], assertion-based software testing searches for a program input data that may lead to the violation of a given assertion. In order to test whether this input data will violate the given assertion or not, assertion-based testing executes the whole program based on based on the given input data. The process of executing the whole program for every assertion may be very costly in larger programs with possibly very large number of assertions. Therefore, the performance of assertion-based software testing may be degraded. In order to alleviate this problem and to enhance the performance of assertion-based software testing in the presence of larger number of assertions, the main goal of this paper is to utilize the advantages offered by testability transformation (TeTra) techniques [8] during the process of assertion-based software testing.

The approach presented in this paper applies testability transformation techniques [8] on an original program $P_o$ with assertions to produce a new version $P_n$ such that assertion-based software testing will be more effective in testing the new version $P_n$ than it would be in testing the old version $P_o$. The primary contributions of this paper are: (1) It introduces a new testability transformation mechanism for programs with assertions. (2) It empowers assertion-based software testing approach and makes more effective in large commercial software with very large number of assertions. (3) The approach may be generally applied to programs with complex pre/post conditions or temporarily embedded pieces of code during instrumentation.

The rest of this paper is organized as follows. A background of assertion-based software testing is presented in Section II. In Section III, related work is discussed. The proposed approach is presented in Section IV. A case study to demonstrate the proposed approach is presented in Section V. Conclusions and future work is discussed in Section VI.

## II. Assertion-Based Software Testing

Assertions have been recognized as a powerful tool for automatic run-time detection of software errors during testing, debugging, and maintenance [9]-[14]. An assertion specifies a constraint that applies to some state of a computation. When an assertion evaluates to a false during program execution, there exist an incorrect state in the program. An approach which employs program assertions for the purpose of test data generation was presented in [9]. In that research, it was shown that assertion-based testing was able to uncover program faults which were uncovered by black-box and white-box testing. Given an assertion *A*, the goal of Assertion-Based testing is to identify program input for which *A* will be violated. The main aim of Assertion-Based software testing is to increase the developer confidence in the software under test. Therefore, Assertion-Based software is intended to be used as an extra and complimentary step *after* all traditional testing methods have been performed to the software. Assertion-Based Testing gives the tester the chance to think deeply about the software under test and to locate positions in the software that are very important with regard to the functionality of the software. After locating those important locations, assertions are added to guard against possible errors with regard to the functionality performed in these locations

An assertion may be described as a Boolean formula built from the logical expressions and from the (**and**, **or**, **not**) operators. There are two types of logical expressions: Boolean expression and relational expression. A Boolean expression involves Boolean variables and has the following form: e1 *op* e2, where e1 and e2 are Boolean variables or true/false constant, and *op* is one of $\{=, \neq\}$. Relational expressions, on the other hand, have the following form: e1 *op* e2, where e1 and e2 are arithmetic expressions and *op* is one of $\{<, \leq, >, \geq, =, \neq\}$. For example, $(x < y)$ is a relational expression, and (f = false) is a Boolean expression.

The goal of assertion-based test data generation [9] is to identify program input on which an assertion(s) is violated. Assertion-based testing is based on goal-oriented testing [2][15], which requires the execution of the program during the process of test data generation. 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 [15]. 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][20] is associated with this statement and function minimization search algorithms are used to find automatically input to execute s. If the search process can not find program input to execute s, this method identifies program's statements that have to be executed prior to reaching the target statement s. 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 [20]. As presented in [9], each assertion 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. Figure 1 shows a sample program with two assertions $A_1$ and $A_2$.

```
program example;
var data: array[1..40] of integer;
var x, i, MAX: integer;
var positive: boolean;
begin
1  input(i, MAX, x);
2  positive:= true;
3  data[i]:= x;
4  while i <= MAX do begin
5    Input(x);
6    i:=i+1;
7    data[i]:= x;
8     if (x ≥ 0) then  begin
9        value:= data[i];
10       write('Value entered: ', value);
      end
      else
      begin
11      value := data[i];
12      write('Value entered: ', value);
13       i:= i-1;
14       positive:= false;
      end;
  (*@  (i ≥1) and (i ≤ 40)  @*)                A1

15    if ((x<0) OR (i=MAX)) AND ((i=MAX)
              OR (positive=false)) then
        begin
16       write(i, MAX, positive);
17       if (i=MAX) OR (positive=false) then
         begin
 (*@ ((i=MAX) or (positive=false)) @*)     A2
18, 19      if (i=MAX) then writeln('Full capacity reached!')
20          else writeln('Negative value entered!');
          end;
       end;
21     positive:= true;
      end;
   end.
```

Figure 1. A Sample program with assertions

Assertion-based software testing [9]-[10] is a promising approach in terms of finding programming bugs. However, this approach may be expensive in terms of search time required to violate each assertion imbedded in the program. This is because this approach is an execution-based approach [2], which depends on finding a program input data that may lead to the violation of an assertion during the program execution on this input data. The problem arises in big size programs with large number of assertions, where the process of re-executing the program for each assertion may be very costly. In order to make assertion-based software testing [9] more effective and efficient in testing big programs with large number of assertions, we propose applying testability transformation [8] on programs with assertions prior to the process of assertion-based software testing.

## III. RELATED WORK

Testability transformation (TeTra) is a source-to-source program code transformation with the objective to make the new programs easier to test [8]. In other words, testability transformation seeks to improve the process of test data generation and makes it more successful. Testability transformation approaches have been applied on many types of programs with encouraging results. For example, in [17], testability transformation improved the performance of Evolutionary Testing (ET) [18] for state-based programs. Korel et al. [19] presented a testability transformation mechanism that is based on data dependencies analysis. In this approach a transformation function is constructed for those program statements that need to be considered during test data generation. Then, the process of test data generation is performed on this transformation function instead of the original program. Although the testability approaches presented in [17] and [19] work well for single program statements they cannot be applied directly for programs with assertions because assertion each assertion may be comprised of more than one program statements as will be shown later in the next section. In order for the approach presented by Korel et al. [19] to be applied on programs with assertions, we need to perform a testability transformation for each assertion found in the program.

## IV. THE PROPOSED APPROACH

The main objective of this paper is to present a testability transformation mechanism for programs with assertions that may makes assertion-based testing more cost-effective and efficient when applied on programs with large number of assertions. Given an original version of a program, $P_o$, with assertions, the proposed approach works as follows.

At the first stage, this approach performs a pre-processing scan of $P_o$ during which all assertions are identified. At the next stage the approach performs a testability transformation process for each assertion identified at the first stage. The results of this stage is that each assertions is transformed into a set of nodes (program statements), as will be explained later, in such a way that executing any of these nodes is equivalent to the violation of this specific assertion. Then, the proposed approach designates each node as a *target* node and formulates a conditional branch (p,q) and a real valued fitness function associated with this branch [2] such that the execution of node p leads to the execution of the target node.

At this stage the chaining approach presented by Ferguson and Korel [16] is employed during the process of assertion-based test data generation to change the program's flow of execution to lead to branch (p,q) such that target node is executed. Because re-executing the original program, $P_o$, during the process of assertion-based test data generation [9] is very costly during the attempt to execute target nodes, in the fourth stage, the proposed approach applies the testability transformation presented in [19] on each of the target nodes as follows.

For each branch (p,q) that leads to the execution of a target node, this testability transformation approach [19] uses data dependency analysis [15][20] in order to identify other program statements that may have influence on leading the program flow towards the target node. There exists a data dependency between two program nodes $n_j$ and $n_k$ with respect to a variable $v$ if the following three conditions are satisfied: (1) $v$ is assigned a valued at $n_j$, (2) $v$ is used at $n_k$, and (3) there exists a program's execution path from node $n_j$ to node $n_k$ where variable $v$ is not modified.

For each of the target nodes identified in the previous stage, the testability transformation approach [19] constructs a data dependency sub-graph [19] and then based on this sub-graph, only selected nodes of the original program, $P_o$, is included in a new code sub-routine called the transformation function: TransFunc() [19]. At this stage, the process of assertion-based test data generation is only performed on TransFunc() in order to find program input data to cause the execution of the associated target node under consideration. By doing so, a huge amount of time is saved during the assertion-based test data generation, because re-executing the TransFunc() is much cheaper than re-executing the whole original program $P_o$ in order to find input program data to execute each target node. Furthermore, it has been shown in [20] that using this method of testability transformation empowers the process of test data generation and makes it more efficient.

In order to clarify how the proposed approach works, consider the following classification. Let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ be a set of assertions found in an original version of a program $P_o$. For each assertion $A \in \mathcal{A}$, a set of nodes $N(A) = \{n_1, n_2, \ldots, n_q\}$ where $q \geq 1$, is identified during a preprocessing stage of the program under test, where the execution of any node $n_k \in N(A)$, $1 \leq k \leq q$, 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 $n_k \in N(A)$ is executed. For example, consider the following sample assertion:

(*@ ((x≥y) **or** (x=z)) **and** ((z≠99) **or** (Full=False)) **and** (z≠0) @*)

The set of nodes for this assertion is: $N(A) = \{ n_1, n_2, n_3 \}$ and the code generated is shown in Figure 2.

```
         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;
```

Figure 2. Code generated for a sample assertion A

In order for an assertion *A* to be violated the search process attempts to generate a program input data **x** that may leads to the execution of *at least* one of $n_1$, $n_2$, or $n_3$.

## V. CASE STUDY

To demonstrate how our proposed approach works, consider assertion $A_1$ in the sample program of Figure 1. In the preprocessing step, assertion $A_1$ is transformed into the following code:

```
p1  IF  i <1 THEN
n11   write('Assertion A1 Violation!');
p2  IF  i > 40 THEN
n22    write('Assertion A1 Violation!');
```

where nodes $n_{11}$ and $n_{22}$ are the constituents nodes for assertion $A_1$ such that the execution of either of these nodes causes the violation of this assertion. Now, the objective of assertion-based testing is to generate program input data that causes the execution of at least one of these nodes [9].

```
function TransFunc(in p_size, int st_ids[], int repts[]): real;
var k, j, i of integer;
var x, i, MAX: integer;
begin
   k:=1;
  while k <= p_size do begin
  case (st_ids[k]) of
        1: begin                              { node 1 }
              input(i, MAX, x);
             break;
           end;
      6: begin                               { node 6 }
             i:= i+1;
            for j:=1 to repts[i]-1 do i:=i+1;
             break;
          end;
        13: begin                            { node 13 }
              i:=i-1;
              for j:=1 to repts[i]-1 do i:=i-11;
              break;
            end;
     end;   { case }
   i:=i+1;
   end;  {while}
   TransFunc:=   (1-i); {return fitness function of problem
node}
end; { function }
```

Figure 3. Testability transformation code generated for assertion A1 to replace original program in Figure 1

In order to lead the program's execution flow towards nodes $n_{11}$ and $n_{22}$, nodes $p_1$ and $p_2$ are designated by the proposed approach as *problem* nodes [19]. In order make the process of assertion-based test data generation more efficient, and to avoid re-executing the whole program, the proposed approach applies data dependency based testability transformation approach [19] on the problem nodes $p_1$ and $p_2$. For example, the testability transformation code generated for the purpose of generating test data to violate assertion $A_1$ through the execution of node $n_{11}$ is shown in Figure 3. Note that the code in Figure 3, only includes program statements that has data dependencies [19] with the problem node $p_1$ with respect to variable *i* which is used at $p_1$. Also, note that the fitness function constructed for the problem node $p_1$ is placed at the return statement of TransFunc() [19] in Figure 3.

By applying this method of testability transformation, only small part of the program code is executed during the process of assertion-based testing which makes assertion-based testing more efficient and suitable for programs with large number of assertions. For example, only the code in shown in Figure 3 is executed during the process applying assertion-based testing on node $n_{11}$ of assertion $A_1$.

## VI. CONCLUSTIONS AND FUTURE WORK

In this paper, we presented a novel software testability transformation for programs with assertions. The presented approach builds upon previous methods of testability transformations and utilizes them for the purpose of making assertion-based testing more efficient. The results of applying the proposed approach on programs with large number of assertions may save valuable testing resources during the process of software testing which enhances rapid development of software products. For our future research, we intend to perform an experimental study to evaluate the effectiveness of the proposed approach in various types of commercial software which may contain large number of assertions.

## REFERENCES

[1] G. Myers, "The Art of Software Testing," John Wiley & Sons, New York, 1979.

[2] B. Korel, "Automated Test Data Generation," IEEE Transactions on Software Engineering, vol. 16, no. 8, 1990, pp. 870-879.

[3] X. Xiaojun and S. Jinhua, "The Study on an Intelligent General-Purpose Automated Software Testing Suite," Intelligent Computation Technology and Automation (ICICTA) International Conference, May 2010, pp. 993-996.

[4] K. Karnavel, K. and J. Santhoshkumar, "Automated software testing for application maintenance by using bee colony optimization algorithms (BCO)," Information Communication and Embedded Systems (ICICES) International Conference, Feb. 2013, pp. 327-330.

[5] P. Srivastava, and K. Baby, "Automated Software Testing Using Metahurestic Technique Based on an Ant Colony Optimization," Electronic System Design (ISED) International Symposium, Dec. 2010, pp. 235-240.

[6] P. Mitra, S. Chatterjee, and N. Ali, "Graphical analysis of MC/DC using automated software testing," Electronics Computer Technology (ICECT) 3rd International Conference, April 2011, pp. 145-149.

[7] D. Rafi, K. Moses, K. Petersen, and M. Mantyla, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," Automation of Software Test (AST) 7th International Workshop, June 2012, pp. 36-42.

[8] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," Software Engineering, IEEE Transactions on, vol. 30, 2004, pp. 3-16.

[9] B. Korel and A. Al-Yami, “Assertion-Oriented Automated Test Data Generation,” Proc. 18th Intern. Conference on Software Eng., Berlin, Germany, 1996, pp. 71-80.

[10] A. Alakeel and M .Mhashi,"Application of Intelligent Assertion-Based Testing in String Matching Algorithms," American Journal of Scientific Research, No. 65, June 2012, pp. 77-91.

[11] D. Rosenblum, "A Practical Approach to Programming With Assertions," IEEE Trans. on Sofware Eng., vol. 21, no. 1, January 1995.pp. 19-31.

[12] K. Shrestha and M. Rutherfor, "An Empirical Evaluation of Assertions as Oracles," Proceedings of IEEE Inter. Conference on Software Testing, Verification and Validation, 2011, pp. 110-119.

[13] S. Khalid, J. Zimmermann, D. Corney, and C. Fidge, "Automatic Generation of Assertiosn to Detect Potential Security Vulnerabilities in C Program That Use Union and Pointer Types," Proceedings of Fourth Inter. Conference on Network and System Security, 2010, pp. 351-356.

[14] A. Alakeel, “Intelligent Assertions Placement Scheme for String Search Algorithms,” Proceedings of the Second International Conference on Intelligent Systems and Applications, Venice, Italy, April 2013, pp. 122-128.

[15] B. Korel, “Dynamic Method for Software Test Data Generation,” Journal of Software Testing, Verification, and Reliability, vol. 2, 1992, pp. 203-213.

[16] R. Ferguson, R. and B. Korel, “Chaining Approach for Automated Test Data Generation,” ACM Tran. on Software Eng. and Tethodology, vol. 5, no. 1, 1996, pp. 63-68.

[17] A. Kalaji , R. Hierons, and S. Swift, “A Testability Transformation Approach for State-Based Programs,” In IEEE 1st International Symposium on Search Based Software, Windsor, UK , May 2009, pp. 85-88.

[18] P. Mcminn and M. Holcombo, “The State Problem for Evolutionary Testing,” Proc. Genetic and Evolutionary Computation Conference, 2003, pp. 2488-2498.

[19] B. Korel, M. Harman, S. Chung, and P. Apirukvorapinit, “Data dependence based testability transformation in automated test generation,” In 16th International Symposium on Software Reliability Engineering (ISSRE 05), Chicago, USA, Nov. 2005, pp. 245–254.

[20] M. Pezze and M. Young, "Software Testing and Analysis: Process, Principles and Techniques," John Wiley & Sons, 2008.

[21] K. Arnold, J. Gosling, and D. Holmes, “The Java programming language,” vol. 2. Reading: Addison-wesley, 1996.

[22] L. Wall, T. Christiansen and J. Orwant, "Programming Perl,", 3rd Ed., O'Reilly Media, 2000.