Using Concurrency to Improve Assertions-Based Software Testing

Ali M. Alakeel
Department of Computer Science
Faculty of Computers and Information Technology
Tabuk University
Tabuk, Saudi Arabia
alakeel@ut.edu.sa

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 assertion processing for which program's concurrency methods are utilized to improve the time required for assertion processing. Using the proposed concurrency model may reduce the time required to explore a large number of assertions during the process of Assertion-Based Software Testing, therefore, making this form of testing more efficient and applicable for programs with large number of assertions.**

Keywords- Assertion-based testing; concurrency,; program assertions; software testing

I. INTRODUCTION

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 [8], [14], [17], [23]. An assertion specifies a constraint that applies to some state of a computation. When an assertion evaluates as 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], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], and can increase program's testability [13], [14], [30].

Utilizing assertions for the purpose of test data generation was proposed in [6]. In this form of testing, 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.

Programming concurrency and parallel processing approaches have been widely used to enhance the performance of software programs related activities, e.g., [27], [28], [29], [30], [31]. However, most of the concurrency work related to software testing has been concentrated in developing testing methods for concurrent programs, e.g., [28]. [29], and very small work has been reported in literature about using program concurrency to speed up the process of software testing for traditional programs with assertions.

This paper presents a new approach for assertion processing for which program's concurrency methods are utilized to improve the time required during Assertion-Based Software Testing (ABST). Since the number of assertions may be very large for industry-size programs, one of the main concerns to the applicability of Assertion-Based Software Testing is the amount of search time required to explore a large number of assertions.

Our proposed approach is based on performing an assertion analyzes process which converts each program assertion into a set of nodes, such that each node is able to execute independently. For nodes execution, we may use a concurrent programming model, e.g., [26], [27] or a parallel processing model, e.g., [31]. Using the proposed concurrency model may reduce the time required to explore a large number of assertions during the process of Assertion-Based Software Testing, therefore, making this form of testing more efficient and applicable for 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 the proposed approach for using program concurrency for assertions processing and section IV, discusses 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], [8], [9], [10], [11], [18], [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], [13], [14], [15], [16], [17], [21] is associated with this statement and function minimization 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].

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. Figure 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: A1 *op* A2, where A1 and A2 are Boolean variables or *true*/*false* constant, and *op* is one of {=, ≠}. On the other hand, relational expression has the following form: A1 *op* A2, where A1 and A2 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.

```
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.
```

Figure 1. A sample program with assertions.

III. Using Program Concuurrency For Assertion Processing

This paper presents a program concurrency approach for assertion processing. The main objective of this approach is to decompose each assertion into simple elements (nodes) where the execution of any of these nodes corresponds to the violation of the associated assertion. Consequently, the process of finding test data to execute each assertion's node is then performed concurrently, therefore saving the time required to explore each assertion. It is important to note that the proposed method in this paper takes a different approach for the optimization of assertion-based testing than that taken by [25].

The proposed concurrent approach does not perform any analysis among assertions and during its operation explores *every* assertion/node found in the program under test, hence avoiding the *elimination* risk associated with the approach presented in [26].

A. Formal Presentation

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

We will show that each program assertion A_i of this form may be converted into a corresponding disjunctive normal form (DNF). Let us denote the DNF of an assertion A_i as DNF(A_i). Furthermore, each DNF(A_i) may be pragmatically represented by a series of if-then statements. For example, consider the following assertion A_1:

(x < y) ***and*** (((z = v) ***and not*** (x > z)) ***or not*** (f = false))

Assertion (A_1) may be converted into a disjunctive normal form as follows. First, we take the negation of A_1, which is:

not ((x < y) ***and*** (((z = v) ***and not*** (x > z)) ***or not*** (f = false)))

This may be simplified to give the form:

not (x<y) ***or*** ((***not*** (z=v) ***or*** (x>z)) ***and*** (f=false))

Which has the following disjunctive normal form:

not (x<y) ***or*** (*not*(z=v) ***and*** (f=false)) ***or*** ((x>z) ***and*** (f=false))

This formula may further be simplified by eliminating ***not*** operands to give the final disjunctive normal form of assertion A_1, DNF(A_1), as shown in Figure 2, below.

(x ≥ y) ***or***
((z ≠ v) and (f = false)) ***or***
((x > z) and (f = false))

Figure 2. Disjunctive normal form of an example assertion A_1.

Eventually each ***or*** clause of DNF(A_1) will form the required predicate conditions to create assertion A_1 constituents nodes expressed as N(A_1). In the case of A_1 there are three nodes, therefore, N(A_1)= { n_1, n_2, , n_3}. Predicate conditions of assertion A_1 nodes are expressed as follows.

C(n_1) = < (x ≥ y)>
C(n_2) = < (z ≠ v), (f = false) >
C(n_3) = < (x > z), (f = false)>

Each node of N(A_1) will represent a single program's statement. In order for this statement to be executed, the associated predicate conditions have to be satisfied. Eventually, assertion A_1 will be replaced a piece of code as shown in Figure 3. In order for assertion A_1 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
n1           ReportViolation_ A1;

         IF  (z ≠ v) THEN
              IF  (f = false) THEN
n2           ReportViolation_ A1;

         IF  (x > z) THEN
             IF  (f = false) THEN
n3           ReportViolation_ A1;
```

Figure 3. Generated code for assertion A_1.

Given a program P with a set of assertions A = {A_1, A_2, …, A_n} with an associated N(A_i) for each assertion A_i, our proposed program concurrency approach for assertion processing proceeds as follows. (1) All nodes $n_k \in$ N(A_i) of assertion A_i have an *equal* chance to be explored during test data generation; (2) the process of generating test data to execute nodes $n_k \in$ N(A_i) is performed *concurrently*. This concurrency can be realized either in a parallel computer environment or in a single computer environment using a programming language that supports multithreading, e.g., Hyper-Threading Technology [26]. Hyper-Threading technology creates two logical processors in a single chip environment which gives the feeling of a real multiprocessor system while running on a single processor environment.

B. The Implementation

This section presents our implantation model for the proposed approach of using programming concurrency in order to enhance the performance of Assertion-Based Software Testing (ABST). It should be noted that the proposed approach may be implemented differently by other software developers based on the computing resources and environment they chose, e.g., [28], [29], [30]. For example, the proposed approach may be implemented using parallel computing environment, e.g N-Cube, or a Mesh architecture. In this text, we present an implementation model that's based on desktop Microsoft Windows architecture using the Java langue concurrency capabilities [27].

As shown in Fig. 3, our implementation model consist of several stages, such that each is responsible for specific operations. The information exchange between these implementation units is achieved through well defended programming interfaces and through data files.

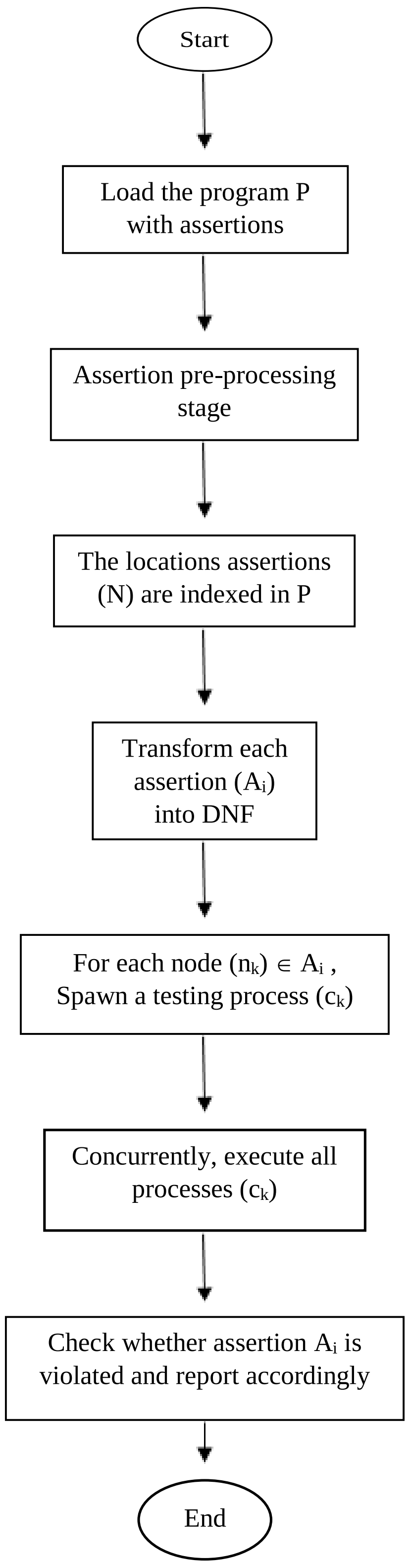
Start
Load the program P with assertions
Assertion pre-processing stage
The locations assertions (N) are indexed in P
Transform each assertion (A_i) into DNF
For each node (n_k) $\in A_i$, Spawn a testing process (c_k)
Concurrently, execute all processes (c_k)
Check whether assertion A_i is violated and report accordingly
End

Figure 4. An implementaion model for the propsed councrrent Assertion-Based Testing.

The implementation model starts by loading the program under test in the testing environment. The program P is then searched for assertions identification process during an assertion preprocessing stage. As an output of this preprocessing stage, the location of each assertion present in the program P is known. At this time, the process of transforming each assertion into the corresponding disjunctive normal form (DNF) is performed as described previously. Note that as a result of this stage, each assertion (A_i) is represented with a set of nodes, N(A_i).

After each assertion *nodes* are identified and represented in DNF, the process of concurrent Assertion-Based Software Testing begins. Remember that the main goal Assertion-Based Software Testing is to produce program's input data to violate each assertion (A_i) found in the program, P. Therefore, a concurrent process is created to execute each node $n_k \in N(A_i)$. Of course the implementation environment has different programming units to ensure the synchronization between running processes and to ensure that the result of each process is reported properly.

IV. Conclusitnos And Future Research

This paper presented an approach for assertions' processing using concurrency. Given a program *P* with assertions, the proposed approach analyses each assertion found in *P* and generates a set of elements (nodes) that represent this given assertion, and then proceeds to explore the processing of each node concurrently. By doing so, the time required to explore each assertion is reduced using this form of concurrency, therefore, making Assertion-Based Software Testing more efficient and applicable in for large programs with thousands of assertions. For our future research, we intend to conduct more experiments to evaluate the performance of the proposed approach on different set of software applications with very 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.

[25] Ali M. Alakeel, “An Algorithm for Efficient Assertions-Based Test Data Generation,” Journal of Software, vol. 5, no. 6, June 2010, pp. 644-653.

[26] https://www.intel.com/content/www/us/en/products/processors/core/i7-processors.html [2018].

[27] B. Goetz, et al., “Java Concurrency in Practice,” Addison-Wesley Professional, 2006.

[28] Fu, H., Wang, Z., Chen, X. et al, “A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques,” Software Quality Journal, No. 26, 2018, pp.855–889.

[29] “How to test your concurrent software: an approach for the selection of testing techniques,” Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems, Vancouver, Canada da, pp. 42-43, 2017.

[30] Guodong Qu, Song-Tao Guo, and Hongqun Zhang, “A practical approach to assertion testing framework based on inner class, IEEE 2nd International Conference on Software Engineering and Service Science, pp. 133-137, 2011.

[31] David B. Kirk and Wen-mei W. Hwu, “Programming Massively Parallel Processors: A Hands-on Approach 2nd Edition,” Publisher: Morgan Kaufmann, 2012.

AUTHORS PROFILE

Ali M. Alakeel, obtained his PhD degree in computer science from Illinois Institute of Technology, Chicago, USA in Dec. 1996, his M.S. degree in computer science from University of Western Michigan, Kalamazoo, USA in Dec. 1992 and his B.Sc. degree in computer science from King Saud University, Riyadh, Saudi Arabia in Dec. 1987. He is currently an Associate Professor of Computer Science at the University of Tabuk, Tabuk, Saudi Arabia. His current research interests include automated software testing, distributed computing and fuzzy logic.