An Algorithm for Efficient Assertions-Based Test Data Generation

Ali M. Alakeel
College of Telecomm & Electronics
Computer Technology Department
Jeddah, Saudi Arabia
Email: dralyami@tvtc.gov.sa

***Abstract*[1]—Automated assertion-based test data generation 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 complex programs, one of the main concerns to the applicability of assertion-based testing is the amount of search time required to explore a potentially large number of assertions. Since assertion-based test data generation is meant to be used *after* programs have been tested using regular testing methods, e.g. black-box and white box, it is expected that most faults have been removed previously, therefore, a large number of assertions will not be violated. If the number of unpromising assertions can be reduced, then the efficiency of assertion-based test data generation can be significantly improved. This paper presents an algorithm which uses data-dependency analysis among assertions in order to accumulate historical data about previously explored assertions which can then be utilized during future explorations. The results of a small experimental evaluation of this algorithm show that the algorithm may reduce the number of assertions to be explored, hence making assertion-based test data generation more efficient. This improvement my vary depending on the number and relationship among assertions found in each program. For example, in a program named MinMax2 with 5 assertions, there was no improvement while in another program named GCD with 24 assertions, there was more than 50% reduction in number of assertions to be explored.**

***Index Terms*—automated software testing, test data generation, software testing, assertion-based testing, program assertions**

I. INTRODUCTION

Software testing is a costly and labor-intensive activity. For this reason, great efforts have been devoted to produce automated testing tools to assist in generating test cases. Given the program under test and a set, I, of its input variables, automatic test data generation is the process of finding input values for I in order to reach a given criterion. Some criteria include statement coverage, branch coverage, and path coverage.

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 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.

As the number of assertions might be very large for complex programs, especially those assertions which are generated automatically, one of the main concerns in the applicability of assertion-based testing presented in [6] is the amount of search time required to explore a potentially large number of assertions in the program under test. Since assertion-based test data generation is meant to be used *after* programs have been tested using

[1] Manuscript received July 7, 2009; revised October 11, 2009; accepted October 31, 2009.

doi:10.4304/jsw.5.6.644-653

regular testing methods, e.g. black-box and white box, it is expected that most faults have been removed previously by these methods. Therefore a large number of assertions will not be violated. If the number of these unpromising assertions can be reduced then the efficiency of assertion-based test data generation can be significantly improved.

This paper presents an algorithm which uses data-dependency analysis among assertions with the intent to accumulate history information about previously explored assertions to be utilized during future explorations. Our main objective is to reduce the time spent during assertions-based testing, hence making this approach more efficient and applicable for complex programs with a large number of assertions. We have implemented this algorithm and used the assertion-based testing method reported in [6] to generate program input to violate a given assertion.

Our experimental evaluation, discussed in Sec. IV, shows that our proposed algorithm, while preserving violation capability, reduced the number of assertions to be explored which lead to less time spent during assertion-based testing. This improvement is not guaranteed for all programs and my vary depending on the number and the relationship among assertions found in each program. The main intent of this experiment is to show that information pertaining to relationships among assertions present in a program can be utilized for the purpose of eliminating some of these assertions during assertions-based testing.

The rest of this paper is organized as follows. Section II provides an overview of assertion-based test data generation. Section III presents our proposed algorithm for efficient assertion-based testing. In Section IV we present our experimental evaluation, and in Section V we discuss our conclusions and future research.

II. ASSERTION-BASED TEST DATA GENERATION

The goal of assertion-based test data generation [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 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.

A. Assertions as Boolean Formulas

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.

B. Assertions as Executable Code

Although most assertions may be described as Boolean formula, a large number of assertions cannot be described in this way. Therefore, our system supports assertions as executable code. The major advantage of "Assertions as executable code" is the flexibility it provides programmers to design as complex assertions as they wish. Assertions in this format are declared in a similar way as Pascal functions that return Boolean value. Local variables may also be declared within an assertion (exactly the same way as in a Pascal function declaration). A special variable *assert* is introduced in each assertion. During assertion evaluation *true*/*false* value has to be assigned to variable *assert*. A sample assertion A2 as executable code is presented in Figure 1. In this assertion variable *j* is a local variable of A2 and all the remaining variables used in A2 are program's variables. The preprocessor translates an assertion into the corresponding function declaration together with the function call in an if-statement. In this paper we are concerned with Boolean-formulas assertions. Therefore, executable code assertions will not be discussed any further.

```
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 two assertions (assertions are shown in italic).

III. ALGORITHM FOR EFFICIENT ASSERTION-BASED TESTING

In our implantation, each program assertion A may be replaced by a block of conditional statements as in Figure 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;
```

Figure 2. Representative code of an assertion A

Formally, let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ be a set of assertions found in a program P. 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. Furthermore, with each node $n_k \in N(A)$ we associate a sequence of nested-if conditions $C(n_k) = < c_1, c_2, \ldots, c_r >$ 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.

For example, Figure 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,
$N(A) = \{ n_1, n_2, n_3 \}$,
$C(n_1) = < (x < y), (x \neq z) >$,
$C(n_2) = < (z = 99), (Full = True) >$, 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;
```

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

Figure 4 shows the corresponding pseudo-code for the algorithm used in [6]. This algorithm processes all assertions independently. Let us refer to this algorithm as ExploreAll.

```
Input:  (𝒜, L)
𝒜: a set of assertions in a program P under test
L: Search time limit
Let StartTime = CurrentTime
WHILE (CurrentTimie –StartTime)<L) DO:
  FOR every assertion A ∈ 𝒜 do:
    FOR every node nk ∈ N(A) do:
      Search for a program input x to execute nk
      IF  x is found THEN
            Report the violation of assertion A;
            Exit For loop
    EndFOR
  EndFOR.
END WHILE
```

Figure 4. A pseudo-code for ExploreAll algorithm

When an assertion is selected for processing by ExploreAll, all nodes for this assertion are processed regardless of the outcome of previously processed nodes. For example, consider assertion *A* defined previously. ExploreAll will attempt to find input data to execute all nodes n_1, n_2 and n_3 regardless of the outcome of previously processed assertions or nodes.

As opposed to the ExploreAll algorithm, our proposed algorithm, ExploreSelect, collects data-dependency information after each exploration of an assertion. This information is then analyzed and used to weed out some unpromising assertion's nodes and may even prevent the exploration of a certain assertion altogether. As shown in Figure 5, ExploreSelect algorithm loops over the set R of assertions to be explored until (i) all assertions in R are explored *or* (ii) the time allowed for assertions processing expires.

```
Input: (𝒜, L)
  𝒜: a set of assertions in a program P under test
  L  : Search time limit
Temporary variables:
  A : current assertion under consideration
  n_k : current node under consideration
  R = ∅, a set of assertions to be explored
  StartTime : temp. var.  holding the time search started
Let R = 𝒜
Let StartTime = CurrentTime
WHILE (R ≠ ∅ )and ((CurrentTime-StartTime )<L)
DO:
     Select[2] next assertion A from R
     WHILE (N(A) ≠ ∅) DO:
        Select[3] next node n_k from N(A)
        Search for a program input x to execute n_k
        IF x is found THEN invoke AnalyzeIfSuccess
           ELSE  invoke  AnalyzeIfFailure
     EndWHILE
EndWHILE.
```

Figure 5. A pseudo-code for ExploreSelect algorithm

ExploreSelect algorithm analyzes results of previously processed assertions or nodes and then tries to employ this result by reducing the size of the set R, i.e., the number of yet to be explored assertions. If the size of R may be reduced then the time spent for assertion-based testing may be reduced. Depending on the result of the current exploration this algorithm invokes a specialized procedure: *AnalyzeIfSuccess* (AIS) procedure is invoked when the system *succeed* in violating the current assertion while the *AnalyzeIfFailure* (AIF) procedure is invoked when the system fails to find test data to violate the currently explored assertion. These special procedures are discussed next. Note that to generate input data to execute a given node n_k, other execution-based test data generation methods, e.g., [2, 3, 9, 16, 22] may be used to fulfill this step.

A. *AnalyzeIfSuccess (AIS) Procedure*

As shown in Figure 6, the AIS procedure 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, AIS heuristic continues program's execution 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, the AIS 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, the AIS performs data dependency analysis after every program execution in order to identify which assertion nodes should be given priority to be explored first in the next execution. Since the AIS is invoked every time the system is able to generate input data **x** for which an assertion node n_k was executed, the objective of this analysis is to: (i) 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; (ii) 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; and (iii) 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 experimental evaluation, presented in Section IV, shows the proposed algorithm, ExploreSelect, *succeeds* in most cases in finding input data to execute a candidate node n_p, hence violating the assertion that n_p is a part of. In other words, a candidate node n_p has a *greater* chance to be executed more than other nodes because of its relation to a previously executed node n_k. To illustrate this, consider the sample program of Figure 7 and its augmented version shown in Figure 8. Notice that the program in Figure 8 is a transformed form of Figure 7's program. Each assertion in Figure 7 has been replaced, in Figure 8, by its corresponding lines of code as explained in Sec. II.

[2] Select statements used in this algorithm are active select, i.e., an item is selected and removed at the same time.
[3] Node selection is based on a priority system which is described in Sec III.A.

```
Input: (A, n_k , R), where
A : an assertion which was violated
n_k : a node n_k ∈ N(A) for which an input data x was found
R : a set of yet to be explored assertions

Report the violation of assertion A
Set N(A) = ∅
Continue program execution on the input x and do the following:
  FOR every executed assertion B DO:
    IF B is violated THEN DO:
        Report the violation of assertion B
        Remove B from R
    EndIF
  EndFOR

After program execution is completed DO:
  FOR every assertion H ∈ R DO:
  FOR every node n_p ∈ N(H) DO:
    IF the following conditions are satisfied:
     1) The conditional sequence of , C(n_p), of n_p
     is identical or a subsequence of the conditional
     sequence, C(n_k ), of n_k ; and
     2) For each variable v ∈ U(C(n_p))[4], v is not
     modified for all paths from n_k to n_p.
    THEN assign node n_p the highest priority to be
          explored next
    EndFOR
  EndFOR
EndDO.
```

Figure 6. A pseudo-code for AnalyzeIfSuccess

With respect to Figure 8, suppose that the system is able to generate the following program input data: i = 15, MAX = 15, and x = 50, for which node 14 ∈ N(A_2) was executed. This means that assertion A_2 is violated. As a reaction to this result, the AIS performs three actions: (1) report the violation of assertion A_2 and remove it from the set R; (2) continue program execution on this input *hoping* that other assertions may also be violated. In this case assertion A_3 (as represented by the nodes 17, 18, 19 and 20 in Figure 8) will also be violated on this input and will be removed from the set R of yet to be explored assertions (notice that assertion A_3 is identical to assertion A_2) and (3) after program execution on this input is completed, all non-violated assertions, i.e. A_4, A_5, A_6, are examined to identify assertion nodes that most likely will be executed when these assertions are explored. Notice that the conditional sequence of node 26 ∈ N(A_4), C(26)= < (i>MAX) >, is identical to the conditional sequence C(14) of node 14 ∈ N(A_2), which has been executed in the current round of execution. This information is recorded and will be used when assertion A_4 is considered for exploration. Specifically, the algorithm uses this information to decide which nodes of A_4 are promising and need to be given a priority over other nodes. To illustrate this, suppose that the system is in a new round of execution and is currently exploring assertion A_4: (*@ (i≥1) and (i≤MAX) @*) defined in Figure 7 and for which the following code was generated in the augmented version as shown in Figure 8:

```
23 IF  i <1 THEN
24     write('Assertion Violation!');
25  IF  i > MAX THEN
26     write('Assertion Violation!');
```

Where N(A_4) = {24, 26},
C(24) = < (i<1) >,
C(26) = < (i>MAX) >.

Using the information collected in the previous round of execution about A_4, and because there exists a program path for which variables *i* and *MAX* used at C(26) are not modified between node 14 and node 26, node 26 has *a great* chance to be executed, i.e., it is most likely that the system will succeed in finding input data for which node 26 will be executed. Therefore, node 26 is assigned a higher priority so that it will be explored before node 24 when assertion A_4 is considered for processing. In connection with node 26, the input data: i = 15, MAX = 15, and x = -1, was what is required to cause the execution of this node, hence the violation of assertion A_4. By examining this input we notice that it only differs in the value of the variable x from that input for which assertions A_2 and A_3 were violated. This implies that, in most situations, it is very *likely* that the system will *succeed* in finding a program input data for which a prioritized node is executed.

B. AnalyzeIfFailure (AIF) Procedure

The AIF procedure, presented in Figure 9, is invoked when the system *fails* to find program input data to execute node n_k ∈ N(A) of a currently processed assertion A. The objective of this algorithm is to identify those *unpromising* conditions (predicates) in the currently processed assertion and to avoid spending valuable search time *repeatedly* trying to find program input data to satisfy these same conditions in case they are part of a yet to be explored assertion' nodes. A condition or a predicate is considered *unpromising* if the system will most likely *fails* to find a program input data to satisfy this condition, i.e., to make this condition evaluate to *true*.

Specifically, given a node n_k ∈ N(A) of a currently explored assertion, for which the system was *not* able to find input data to execute this node, the AIF heuristic identifies condition c ∈ C(n_k) (i.e., c belongs to the conditional sequence C(n_k) of node n_k) which was not satisfied, i.e., did not evaluate to *true*.

[4] The set of used variables for the conditional sequence of a node n_p is defined as follows:

$U(C(n_p)) = \bigcup_{c \in C(n_p)} U(c)$, where $U(c)$ is the set of used variables at a single condition c.

```
program example;
Var data: array[1..40] of integer;
  x, i, MAX: integer;
  positive:boolean;
begin
1  Input(i, MAX, x);
2  positive:= true;
   (*@   (i ≥1) and (i ≤ MAX)   @*)            A1
3  data[i]:= x;
4  while i <= MAX do begin
5    Input(x);
6    i:=i+1;
   (*@   (i ≥1) and (i ≤ MAX)   @*)            A2
7    data[i]:= x;
8     if (x ≥ 0) then  begin
   (*@   (i ≥1) and (i ≤ MAX)   @*)            A3
9         value:= data[i];
10        write('Value entered: ', value);
      end
      else
    begin
  (*@  (i ≥1) and (i ≤ MAX)  @*)               A4
11            value := data[i];
12            write('Value entered: ', value);
13            i:= i-1;
14                    positive:= false;
                  end;
15            if ((x<0) OR (i=MAX)) AND ((i=MAX)
              OR (positive=false)) then
              begin
 (*@ (((x<0) or (i=MAX)) and ((i=MAX) or
              (positive=false)))  @*)          A5
16                  write(i, MAX, positive);
17                  if (i=MAX) OR (positive=false) then
                    begin
 (*@ ((i=MAX) or (positive=false)) @*)         A6
18, 19                if (i=MAX) then writeln('Full
                      capacity reached!')
20                    else writeln('Negative value
entered!');
                end;
              end;
21          positive:= true;
          end;
end.
```

Figure 7. Sample program with repeated assertions

With the condition c on hand, the AIF scans every node $n_p \in N(A) \cup N(H)$, $p \geq 1$, for all assertions $H \in R$ (the set of yet to be explored assertions) looking for any node, n_p, for which $c \in C(n_p)$, i.e., nodes that include condition c as a part of their conditional sequences. For every such node n_p, AIF performs data-dependency analysis to check if any of the variables used in condition $c \in C(n_p)$ was modified between node n_k and node n_p. If this analysis reveals that *none* of the variables used at $c \in C(n_p)$ was modified between node n_k and node n_p then this indicates that it is very *likely* that the system will also *fail* to find input data for which node n_p will be executed, i.e.,

```
program example;
Var       data: array[1..40] of integer; x, i, MAX: integer;
positive:boolean;
begin
1           Input(i, MAX, x);
2           positive:= true;
3, 4        if i<1 then     write('Assertion Violation!');
5, 6        if i>MAX then   write('Assertion Violation!');
7           data[i]:= x;
8           while i <= MAX do
            begin
9             Input(x);
10            i:=i+1;
11, 12      if i<1 then     write('Assertion Violation!');
13, 14        if i>MAX then   write('Assertion Violation!');
15            data[i]:= x;
16            if (x ≥ 0) then
              begin
17, 18         if i<1 then     write('Assertion Violation!');
19, 20         if i>MAX then   write('Assertion Violation!');
21               value:= data[i];
22               write('Value entered: ', value);
              end
              else begin
23, 24        if i<1 then     write('Assertion Violation!');
25, 26         if i>MAX then   write('Assertion Violation!');
27                    value := data[i];
28            write('Value entered: ', value);
29             i:= i-1;
30             positive:= false;
        end;
31      if ((x<0) OR (i=MAX)) AND ((i=MAX)
              OR (positive=false)) then
       begin
32, 33, 34    if x ≥ 0 then if i ≠ MAX  then
                    write('Assertion Violation!');
35, 36, 37    if i ≠ MAX then if positive ≠ false then
                    write('Assertion Violation!');
38      write(i, MAX, positive);
39      if (i=MAX) OR (positive=false) then
            begin
40, 41, 42    if i ≠ MAX then if positive ≠ false then
                    write('Assertion Violation!');
43, 44          if (i=MAX) then writeln('Full capacity
                                        reached!')
45              else writeln('Negative value entered!');
            end;
       end;
46     positive:= true;
    end;
end.
```

Figure 8. Augmented version of program in Figure 7

node n_p has a very small chance to be executed. Therefore, every node n_p is considered as *unpromising* and is removed from the set of nodes to be explored.

Although a removed node might have had a very slight chance to be executed, had it been explored, there is a greater chance that it will *not* be executed as supported by

```
Input: (A, n_k, c, R)
  A: currently explored assertion
  n_k: a node n_k ∈ N(A) for which an input data was not found
  c: the condition of C(n_k) which was not satisfied during exploration of node n_k
  R: a set of yet to be explored assertions

FOR every assertion H ∈ R ∪ {A}DO:
  FOR every node n_p ∈ N(H) DO:
    IF the following conditions are satisfied:
      1) c is part of the conditional sequence of n_p, C(n_p);
      AND
      2) For every variable v ∈ U(c), v is not modified
         for all paths from n_k to n_p
    THEN remove n_p from N(H)
  EndFOR
EndFOR.
```

Figure 9. A pseudo-code for AnalyzeIfFailure

our experimental evaluation. Since the time to explore a single node is expensive and since the objective of the AIF heuristic is to reduce the time consumed in assertion processing, the little risk taken in removing nodes is well justified, especially that this risk is so little if not zero in many cases.

To illustrate how AIF procedure decides *not* to explore *unpromising* nodes and remove them from the list of nodes to be explored during assertion processing, consider the sample program in Figure 7 and its augmented version in Figure 8.

Consider assertion A_5: (*@ (((x<0) or (i=MAX)) and ((i=MAX) or (positive=false))) @*) defined in Figure 7. A_5 was replaced by the following code in the augmented version of that program appeared in Figure 8:

```
32  IF x ≥ 0 THEN
33    IF i ≠ MAX THEN
34      write('Assertion Violation!');
35    IF i ≠ MAX THEN
36      IF positive ≠ false THEN
37        write('Assertion Violation!');
```

Where,
$N(A_5) = \{34, 37\}$,
$C(34) = < (x \geq 0), (i \neq MAX) >$,
$C(37) = < (i \neq MAX), (positive \neq false) >$.

Suppose that node 34 was selected first during the processing of A_5 and that the system was not able to find input data **x** for which the condition (i ≠ MAX) is satisfied. Consequently node 34 is not executed. Based on the outcome of this event, the AIF procedure inspects the remaining nodes of A_5 and explores the possibility of eliminating the processing of some of these nodes. Specifically AIF will do the following:

1) Identify the condition(s) c_f among the conditional sequence of node 34, C(34), through which the execution of node 34 was not possible; and
2) For every node $n_p \in N(A) \cup N(B)$, for all assertions $B \in R$, for which $c_f \in C(n_p)$ remove n_p from the set of nodes to be explored. Notice that other conditions in the same sequence for the same node are dropped as well because they were *anded* together with the failed condition.

In this example, c_f = (i ≠ MAX), is the condition through which the execution of node 34 was not possible. By inspecting node 37 we notice that c_f also belongs to the conditional sequence of node 37, i.e., $c_f \in C(37)$. Based on this finding and because both variables *i* and *MAX* used in the condition (i ≠ MAX) at node 37 were not modified since their last use at node 34, the AIF considers node 37 to be an *unpromising* node and, consequently, will not invest search time trying to execute this node. Node 37 is considered unpromising because it is very *likely* that the system will *fail* to find input data to execute this node as was the case with node 34. This is because it is necessary to satisfy the condition (i ≠ MAX) in order for node 37 to be executed. As supported by our experimental evaluation it is most likely that the system will not be able to find input data to execute these nodes.

As another example to illustrate how AIF eliminates the processing of an assertion based on the result of a previously processed assertion which was *not* violated, consider the following situation. Given the information about assertion A_5 presented in the previous example, consider assertion A_6: (*@ ((i=MAX) or (positive=false)) @*) of Figure 7 which was replaced by the following code in Figure 8:

```
40  IF i ≠ MAX THEN
41  IF positive ≠ false THEN
42    write ('AssertionViolation');
```

For this assertion we have:
$N(A_6) = \{42\}$,
$C(42) = < (i \neq MAX), (positive \neq false) >$.

Recall that assertion A_5, considered in the previous example, was *not* violated because the system was not able to generate input data for which the condition c_f = (i ≠ MAX) will be satisfied. Where $c_f \in C(34)$, the conditional sequence of node 34. Now, inspecting the conditional sequence, C(42), of node $42 \in N(A_6)$ we notice that c_f is also a member of C(42). Based on (i) the fact that the system had previously failed to find input data to satisfy c_f, (ii) since c_f also belongs to C(42), and (iii) variables (i and MAX) used in c_f were not modified since their last use at node 34, the AIF considers node 42 as *unpromising* node and node 42 will not be considered for exploration. Because node 42 is the *only* node in $N(A_6)$ assertion A_6 is removed from the set R of yet to be explored assertions and the time to generate input data to violate this assertion is saved.

IV. EXPERIMENTAL EVALUATION

The intent of this experiment is only to show that information pertaining to relationships among assertions present at a program can be utilized for the purpose of eliminating some of these assertions during assertions-based testing. Results may depend on the number and the relationship among assertions found in each program.

To derive our experiment, a suite of fifteen Pascal programs with assertions was used. In order to evaluate the performance both ExploreSelect (ES) and ExploreAll (EA) algorithms with respect to programs with potential assertion violations and those which might not have any assertion violations, we have used a mix suite of correct and faulty programs. Programs, to be described later, used in this experiment include: Bank, GCD, Bubble, Stack, Prime, MinMax1-MinMax8, Total, and Average. From these programs, GCD, Bubble, Stack and Prime are assumed to be fault-free, to the best of our knowledge, while Bank, MinMax, Total, and Average, have been seeded with at least one fault.

This experiment is performed as follows: each program used in this experiment is tested using assertion-based testing reported in [6] in two rounds: one is using EA algorithm and the other uses ES algorithm. Remember that assertion-based testing as described in [6] is only performed after each program has been tested using both black-box testing and white-box testing (branch coverage). During this experiment, for each program we recorded (1) the total time (in minutes and seconds) consumed by each algorithm to perform the test (i.e., to try to violate assertions found in each program), (2) number of assertions explored by each approach and (3) number of assertions violated by each approach. The complete result of this experiment is presented in TABLE I which entries should be interpreted as follows: Column #1 and Column #2 give the program name and the number of assertions (NA) in this program, respectively. Column #3 shows the total time (in minutes and seconds) required by EA and ES algorithms to explore all assertions in a certain program. Column #4 shows the total number of assertions explored using EA and ES. Finally, Column #5 gives the total number of assertion violations achieved by EA and ES algorithms. For example, the second entry of TABLE I shows (i) that the EA spent approximately three hours and nineteen minutes to explore 24 assertions found in program GCD while the ES spent about an hour to explore the same number of assertions, (ii) that EA explored all the 24 assertions found in this program while ES explored only eleven assertions, and (iii) none of the assertions found in this program were violated by either EA or ES algorithms.

A. The Programs

A brief description of the programs, developed for the purpose of this experiment, will now be given. Program Bank performs simple banking operations such as opening an account and depositing and withdrawing money. Program GCD computes the greatest common divisor of an array of integers. Program Bubble sorts an array of integer using the bubble sort algorithm. Program Stack implements typical stack operations such as push, pop, empty and full. Program Prime finds the set of prime numbers out of a given input integers list. Program MinMax finds the minimum and the maximum of an array of integers (versions 1 to 6 of this program differ in the type and location of the fault seeded, while versions 7 and 8 differ from other versions in the algorithm used to compute the minimum and the maximum). Programs Total and Average compute the total and the average of an array of integers, respectively. Number of uncommented lines of code for the programs is as follows: Bank (336), GCD (177), Bubble (54), Stack (114), Prime (94), MinMax (68), Total (52), and Average (54).

B. Discussion of the Experiment

As shown in TABLE I, by using ES algorithm we were able to reduce the amount of time spent for assertion processing by 56%. This means that by using the ES algorithm, more than half of the time that is consumed by EA algorithm has been spared. This is a significant saving considering the value of time during software testing. The good performance by the ES is mostly attributed to its ability to better invest the search time by eliminating unpromising assertions and/or nodes during assertions processing. In this respect, ES algorithm was able to reduce the number of assertions explored by 17% as shown in the bottom of TABLE II. Although eliminating assertions is not possible for some programs, ES algorithm attempts to eliminate unpromising nodes within assertions which results in reducing the overall time required for assertions processing. This explains why ES algorithm spends less time than EA algorithm to explore the same number of assertions (six assertions) in program Stack (shown in the "Time" column in the fourth entry of TABLE I). The reason for this is that through nodes elimination, ES algorithm was able to eliminate the processing of four out of twelve assertion's nodes found in program Stack. This has reduced the number of nodes to explore in this program to eight while EA algorithm had to explore all twelve nodes (shown in the fourth entry of TABLE II). For all programs in this experiment, ES algorithm reduced the number of assertion's nodes to explore by 37% as reflected in TABLE II. In addition to these improvements by ES over EA algorithm, ES algorithm was able to violate the same number of assertions as EA algorithm, which means that there was no risk incurred by using the ES with respect to the programs used in this experiment. Node elimination raises an important issue. It was discussed previously in Sec III.B, that during assertion processing some unpromising nodes are eliminated by the AIF heuristic.

TABLE I.
EXPERIMENTAL RESULTS

Keys:
NA: Total number of assertions in the program
NE: Number of explored assertions
NV: Number of violated assertions
ES: ExploreSelect algorithm
EA: ExploreAll algorithm

Program	NA	Time (minutes)		NE		NV	
		EA	ES	EA	ES	EA	ES
Bank	19	72.66	49.21	19	17	1	1
GCD	24	191.42	58.49	24	11	0	0
Bubble	4	1.18	0.79	4	3	0	0
Stack	6	3.50	2.34	6	6	0	0
Prime	6	12.15	10.11	**6**	6	0	0
MinMax1	5	0.32	0.33	5	5	2	2
MinMax2	5	0.32	0.32	5	5	2	2
MinMax3	5	0.25	0.22	5	5	4	4
MinMax4	5	0.32	0.30	5	5	3	3
MinMax5	5	0.28	0.30	5	5	3	3
MinMax6	5	0.23	0.22	5	5	4	4
MinMax7	5	0.58	0.39	5	4	1	1
MinMax8	5	0.58	0.39	5	4	1	1
Total	2	0.54	0.5	2	2	1	1
Average	2	0.29	0.3	2	2	1	1
Total	103	284.62	124.21	103	85	23	23
Average	6.87	18.97	8.28	6.87	5.67	1.53	1.53
Reduction by ES			56%		17%		
Elimination's Risk							0%

TABLE II.
NUMBER OF EXPLORED ASSERTION'S NODES

Program	Total No. of Nodes	EA	ES
Bank	35	35	13
GCD	53	53	15
Bubble	12	12	8
Stack	12	12	8
Prime	11	11	11
MinMax1	7	7	7
MinMax2	7	7	7
MinMax3	7	7	7
MinMax4	7	7	7
MinMax5	7	7	7
MinMax6	7	7	7
MinMax7	7	7	5
MinMax8	7	7	5
Total	3	3	3
Average	3	3	3
Total	179	179	113
Reduction By ES		37%	

Because of the nature of the test data generation problem, where it is impossible to test a program for *all* possible inputs [12], some eliminated nodes *may* have some chance in being executed (i.e., results in an assertion violation) had they given the opportunity to be explored.

Although the risk imposed by node elimination is considered a limitation of ES algorithm, the results of our experimental study shows that this risk is minimal where, for all programs used in this study, both EA and ES algorithms were able to violate the same number of assertions (i.e., no risk was incurred by using ES algorithm). Although there is a little risk associated with node elimination by ES algorithm, this risk is a reasonable compromise to take for the speed achieved using this heuristic because (i) this risk is minimal as supported by our experimental evaluation and (ii) eliminating an unpromising node n_p only takes place when the search was not successful in finding input data to violate a *related* node n_k and, in most cases, executing node n_p would *unlikely* lead to the violation of the currently explored assertion.

V. CONCLUSIONS

This paper presents *ExploreSelect*, an algorithm for efficient assertion-based automated test data generation. *ExploreSelect* uses data-dependency analysis among assertions found in the program in order to reduce the

time required for assertion-based test data generation.

Currently, this algorithm is implemented for assertions represented as Boolean formulas. Considering the number assertions of this type may be very large as they are generated automatically, (as they are supported by some programming language), the time required to process such larger number of assertions may hamper the applicability of assertions-based testing for large programs. Examples of such are assertions that guard for array-boundary violations, division by zero, integer/float underflow/overflow, stack overflow, etc. ExploreSelect utilizes data-dependency analysis in eliminating unpromising assertions during a pre-scan process, thereby avoiding wasting valuable search time trying to violate such assertions.

Our experimental evaluation shows that, using ExploreSelect has significantly reduced the time required to perform assertion-based test data generation as compared to ExploreAll algorithm which process all assertions independently. Although ExploreSelect may eliminate some assertion's nodes or decide not to explore a given assertion(s) altogether, our experimental evaluation shows that this process did not diminish its ability in assertion violation nor does it change the program's testability. This is because removed nodes and/or assertions have a very little chance to be violated.; Therefore, ExploreSelect preserves the performance of the ExploreAll in terms of assertion violations. The purpose of this experiment is to show that information among assertions may be utilized during assertion-based testing but does not guarantee the same result for all programs. Improvements may vary depending on the number and the relationship among assertions found in each program. In the future, we plan to perform additional experiments using larger sized programs in order to evaluate the applicability of this algorithm for commercial software.

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] C. Hulten, "Simple Dynamic Assertions for Interactive Program Validation," AFIPS Conference Proceedings, Las Vegas, 1984, pp. 405-410.

[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] J. Voas, "How Assertions Can Increase Test Effectiveness," IEEE Software, March 1997, pp. 118-122.

[14] J. Voas, K. Miller, "Putting Assertions in Their Place," Proceedings of the International Symposium on Software Reliability Engineering, 1994.

[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] L. Stucki, G. Foshee, "New Assertion Concepts for Self-Metric Software Validation," Proceedings of the International Conference on Reliable Software, 1975, pp. 59-71.

[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.

Ali M. Alakeel, also known as Ali M. Al-Yami, obtained his PhD degree in computer science from Illinois Institute of Technology, Chicago, USA on Dec. 1996, his M.S. degree in computer science from University of Western Michigan, Kalamazoo, USA on Dec. 1992 and his B.Sc. degree in computer science from King Saud University, Riyadh, Saudi Arabia on Dec. 1987. He is currently an Assistant Professor of Computer Science at the College of Telecomm & Electronics, Jeddah, Saudi Arabia. His current research interests include automated software testing, fuzzy logic and distributed computing.