Assertion-Oriented Automated Test Data Generation [*]

Bogdan Korel, Ali M. Al-Yami
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
email: korel@charlie.iit.edu

Abstract

Assertions are recognized as a powerful tool for automatic run-time detection of software errors. However, existing testing methods do not use assertions to generate test cases. In this paper we present a novel approach of automated test data generation in which assertions are used to generate test cases. In this approach the goal is to identify test cases on which an assertion is violated. If such a test is found then this test uncovers an error in the program. The problem of finding program input on which an assertion is violated may be reduced to the problem of finding program input on which a selected statement is executed. As a result, the existing methods of automated test data generation for white-box testing may be used to generate tests to violate assertions. The experiments have shown that this approach may significantly improve the chances of finding software errors as compared to the existing methods of test generation.

1. Introduction

Software testing is a very labor-intensive and expensive process. If the process of testing could be automated, significant reductions in the cost of software development could be achieved. Test data generation in software testing is the process of identifying program input which satisfy selected testing criteria. There exist two major classes of methods of test data selection: black-box testing and white-box testing.

* This research was partially supported by the NSF grant CCR-9308895.

In black-box testing test data are selected based on the functional specification of the program (e.g., equivalence class partitioning, boundary-value analysis). In white-box testing test data are selected based on the structure of the program (e.g., statement coverage, branch coverage, etc.). A test data generator is a tool which assists a programmer in the generation of test data for the program. 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, etc. After initial testing, programmers frequently face the problem of finding additional test data to evaluate program elements not yet covered. Finding input test data to evaluate the remaining elements can be very labor-intensive and expensive. Therefore, test data generation in white-box testing is a process of finding program input on which a selected element (e.g., a not yet covered statement) is executed. There are three types of automated test data generators: *random* test data generators [1], *path-oriented* test data generators [3, 4, 19, 12], and *goal-oriented* test data generators [13, 14, 7]. The path oriented approach is the process of selecting a program path(s) to the selected statement and then generating input data for that path. Two methods have been proposed to find program input to execute the selected path, namely, *symbolic execution* [3, 4, 9, 5] and *execution-oriented test data generation* [12]. The goal-oriented approach of test data generation [13, 14, 7] differs from the path-oriented test data generation in that the path selection stage is eliminated. The general idea of the goal-oriented approach is to concentrate only on this part of the program that "affects" the execution of the selected statement, and to ignore these parts of the program that do not influence the execution of this statement. This approach has recently been extended to an approach, referred to as a chaining approach, in which data dependence analysis is used to guide the search process [7].

Assertions are recognized as a powerful tool for automatic run-time detection of software errors during debugging, testing, and maintenance. An assertion specifies a constraint that applies to some state of a computation. When the assertion evaluates to false during program execution, there exists an incorrect state in the program. Assertions proved to be very effective in testing and debugging cycle. For example, during black-box and white-box testing assertions are evaluated for each program execution. Information about assertion violations is used to localize and fix bugs. However, the existing testing methods do not use information about assertions to generate test cases.

In this paper we present an approach of automated test data generation which is centered around assertions, i.e., assertions are used to generate test cases. In this approach the goal of assertion-oriented testing is to identify a program test on which an assertion(s) is violated. If such a test(s) is found then this test uncovers an error in the program. The assertion-oriented test generation is a very attractive approach of test data generation. However, finding program inputs by programmers on which selected assertions are violated might be very expensive and time consuming. Therefore, in order for the assertion-based test generation to be of practical value it is imperative that test data that violate assertions are generated automatically. We will show that the problem of finding program input on which an assertion is violated may be reduced to the problem of finding program input on which a selected statement is executed. As a result, the existing methods of automated test data generation for white-box testing [1, 3, 4, 7, 12, 14, 19] may be used to generate program inputs to violate assertions.

The assertion-oriented test data generation has been developed as a part of the test data generation system TESTGEN [11, 15]. TESTGEN supports test data generation for programs written in a subset of Pascal. Our initial experience with the assertion-oriented test data generation is very encouraging. This approach may efficiently find many faults that are not detected by other testing methods. Since the test generation is fully automatic, this approach does not require much effort on the part of programmers. In addition, unlike the existing testing methods, each test case that is generated by the assertion-oriented approach uncovers an error in the program.

The organization of this paper is as follows: in section 2 an overview of assertions is presented. Section 3 presents the assertion-oriented data generation. In section 4 the test generation in TESTGEN is presented. Section 5 presents the results of an experimental study. In section 6 future research is discussed.

2. Program Assertions

An assertion specifies a constraint that applies to some state of a computation. When the assertion evaluates to false then there exists an incorrect state in the program. The idea of using assertions is not new. The first systems supporting simple assertions were already reported in 1970s [2, 10, 21, 23]. It has been shown that assertions may be very effective [20] in detecting run-time errors. The effort in constructing the assertions pays off in quick and automatic detection of faults. Assertions are supported by some programming languages, e.g., Eiffel [18] and Turing [8]. For languages that do not support assertions specialized tools were developed, e.g., APP tool [20] supports assertions for C programs. These tools recognize assertions that appear as annotations of the programming language. Typically, the assertions are written inside comment regions using the extended comment indicators. A special preprocessor may be used to convert assertions to embedded self-checks that are invoked during program execution.

Effectiveness of assertions is an important issue [17]. Attempts are made to characterize the kinds of assertions that are most effective in uncovering errors, i.e., assertions that most likely uncover program faults and errors. Such assertions should be violated for typical program inputs that are likely to be used during program testing. Rosemblum [20] presented a classification of the assertions that may be effective in detecting faults. Most of these assertions guard against many common kinds of faults or errors. It has been shown that many assertions may be created without major effort.

There are many different ways assertions can be described. In this paper we describe assertions supported in the system TESTGEN [11, 15] that supports test data generation for Pascal programs. Assertions supported by TESTGEN are written inside Pascal comment regions using the extended comment indicators: (*@ assertion @*). In this way assertions may easily be recognized by the preprocessor that substitutes the assertions with self-checks. We distinguish two types of assertions in TESTGEN: Boolean formula or executable code.

Assertion as a Boolean formula:

An assertion may be described as a Boolean formula built from the logical expressions and from the **and**, **or**, **not** operators. In TESTGEN 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 *false/true* constant, and *op* is one of $\{=, \neq\}$. 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 $\{<, \leq, >, \geq, =, \neq\}$. 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 TESTGEN translates assertion A into the following code:

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

When assertion A is violated, a special procedure Report_Violation is called that reports the assertion violation.

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, TESTGEN supports assertions as executable code. The major advantage of assertions in this form is that very complex assertions may be provided by programmers. Assertions in this format are declared in a similar way as Pascal functions that return boolean value. In TESTGEN, 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.

```
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).

3. Assertion-Oriented Test Data Generation

In this section we present a novel approach of automated test data generation in which assertions are used to generate test cases. In this approach the goal is to identify program input on which an assertion(s) is violated. More formally we can define the problem of assertion-oriented test data generation as follows: For a given assertion *A*, find program input ***x*** on which assertion *A* is false, i.e., when the program is executed on input ***x*** and the execution reaches assertion *A*, assertion *A* evaluates to false.

The problem of finding a program input on which an assertion is violated is undecidable. Therefore, it is not always possible to generate test cases that violate assertions. However, the experiments have shown that the assertion-oriented test generation may frequently detect errors in the program related to assertion violation. The major advantage of this approach is that each test data generated uncovers an error in the program because the assertion is violated. There are three reasons as to why an assertion is violated: a faulty program, a faulty assertion, or a faulty precondition (input assertion).

The problem of finding program input on which an assertion is violated can be reduced to the problem of finding program input on which a selected statement (node) is executed. For each assertion special source code is generated by the preprocessor; in this way an augmented version of the program is created. The augmented program contains special statements that are inserted into it in such a way that the execution of one of these statements corresponds to the violation of an assertion(s). As result, it is possible to use the existing methods of test data generation that find program input to execute these special statements in the augmented program. In what follows we describe the augmentation process for both types of assertions:

Assertion as a Boolean formula:

The goal is to find program input on which assertion *A* is evaluated to false, or equivalently the goal is to find program input on which Boolean formula (**not** *A*) evaluates to true. For this purpose for given assertion A we find the simplified disjunctive normal form of (**not** *A*). Notice that each Boolean formula may be represented in disjunctive normal form. In addition, it is possible to simplify the Boolean formula in the disjunctive normal form by eliminating **not** operands by applying the following simple transformations to the logical expressions:

Relational expressions		*Boolean expressions*	
with **not**	without **not**	with **not**	without **not**
not $(a < b)$	$a \geq b$	not $(a = b)$	$a \neq b$
not $(a \leq b)$	$a > b$	not $(a \neq b)$	$a = b$
not $(a > b)$	$a \leq b$		
not $(a \geq b)$	$a < b$		
not $(a = b)$	$a \neq b$		
not $(a \neq b)$	$a = b$		

Let F be the simplified disjunctive normal form of (**not** *A*). For each product (e_1 **and** e_1 **and** ... **and** e_n) of formula F the following nested if-statements are created:

```
if $e_1$ then
    if $e_2$ then
        ......
            if $e_n$ then Report_Violation;
```

If product (e_1 **and** e_1 **and** ... **and** e_n) evaluates to true then assertion *A* is violated. Equivalently, if 'Report_Violation' statement is executed, then assertion *A* is violated.

Example 1:

For the following assertion A:

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

the negation of A

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

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 elimination of **not** operands:

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

For this formula the following code is generated:

```
if x ≥ y then Report_Violation;
if z ≠ v then
        if f = false then Report_Violation;
if x > z then
        if f = false then Report_Violation;
```

Now the goal is to find a program input on which one of the 'Report_Violation' statements is executed. This is equivalent to the violation of the assertion *A*. As a result, we have reduced a problem of finding program input on which an assertion is violated to the problem of finding an input on which a node is executed in the augmented program.

Assertions as executable code:

An assertion as executable code is violated if one of the statements 'assert:=false' inside of the assertion is executed. Since each assertion in this form is transformed into a Pascal function in the augmented program, execution of an assignment statement that corresponds to the 'assert:=false' statement in the assertion declaration constitutes the assertion violation. Therefore, the problem of finding program input on which the assertion is violated is reduced to the problem of finding program input on which a statement is executed (an assignment statement that assigns the false-value to the assertion function identifier) and this false-value is preserved up to the end of the execution of assertion function. For each assertion TESTGEN identifies all such statements and then tries to find program input on which one of these statements is executed.

Example 2:

Figure 2 presents the augmented version of the program of Figure 1. In the augmented program the violation of assertion A1 is equivalent to the execution of statement *k2* or *k4*, whereas the violation of assertion A2 is equivalent to the execution of statement *n5* (these statements are shown in bold).

Automatically generated assertions:

Certain types of assertions may automatically be generated from the source code. Typically, these assertions guard against certain classes of run-time errors (for example, division by zero, array boundary violation, etc.). TESTGEN supports automatic generation of assertions for different classes of possible run-time errors. For example, for the following statement: x := y / (v + z); the following assertion may be automatically generated: (*@ v + z ≠ 0 @*). This assertion guards against the division by zero error. In the program of Figure 1, the assertion A1 guards against the array boundary violation and it may be generated automatically. TESTGEN automatically identifies statements with potential run-time errors and then generates appropriate assertions that are then used in the assertion-oriented test data generation. In the current implementation TESTGEN generates assertions for the following run-errors: division by zero, array boundary violation, overflow error. TESTGEN also supports automated detection of uninitialized variables, i.e., statements at which referenced variable may be uninitialized. Static analysis is used to identify potential uninitialized variables; TESTGEN then tries to identify program inputs on which these variables are uninitialized.

```
program sample;
var
n: integer;
a: array[1..10] of integer;
i,max,min: integer;

{Code generated for assertion A2}
function A1: boolean;
var
j: integer;
begin
n1      A1:=true;
n2      j:=1;
n3      while j ≤ n do begin
n4,n5     if max < a[j] then A1:=false;
n6          j:=j+1;
          end;
end;

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;
              {Code generated for assertion A1}
k1,k2         if i < 1 then Report_Violation;
k3,k4         if i > 10 then Report_Violation;
9,10          if max < a[i] then max:=a[i];
       end;

              {Code generated for assertion A2}
n7,n8         if not A1 then Report_Violation;

11     writeln(min,max);

end.
```

Figure 2. An augmented program of Figure 1 with two assertions (the augmented code is shown in italic).

4. Test Generation in TESTGEN

In TESTGEN test data is generated using the chaining approach [7] that is an extension of execution-oriented methods of test data generation [13, 14]. The test generation problem is defined as follows: Given node *g* (referred to as the goal node) in a program. The objective is to find program input **x** on which node *g* will be executed.

The general idea of the chaining approach is to concentrate only on this part of the program that "affects" the execution of the selected node, and to ignore these parts of the program that do not influence the execution of this statement. The chaining approach uses data dependency analysis to guide the test data generation process. It identifies a sequence of nodes that may affect the execution of the selected node g. The chaining approach uses the notion of last definition to identify such a sequence: By a *last definition* of variable v at node p we mean a node that assigns a value to variable v and this value may potentially be used by node p. For example, node 2 is a last definition of variable *max* at node *n4* in the program of Figure 2.

The chaining approach starts by executing a program for an arbitrary program input **x**. During program execution for each executed branch (p,q), the search process decides whether the execution should continue through this branch or whether an alternative branch should be taken (because, for instance, the current branch does not lead to goal node *g*). If an undesirable execution flow at the current branch (p,q) is observed, then a real-valued function is associated with this branch. Function minimization search algorithms are used to find automatically new input that will change the flow execution at this branch. If, at this point, the search process cannot find program input **x** to change the flow of execution at branch (p,q), then the chaining approach attempts to alter the flow at node p by identifying nodes that have to be executed prior to reaching node p. As a result, the alternative branch at p may be executed. We will refer to node p of the branch (p,q) as a *problem* node. The chaining approach finds a set of last definitions of all variables used at problem node p, in order to identify nodes in the program which have to be executed prior to the execution of problem node p. Such a sequence of nodes to be executed is referred to as an *event sequence* and used to control the execution of the program by the chaining approach.

Event sequences are used to "guide" the execution of the program during the search process. Informally, by an *event* we mean the fact that a node is executed. Similarly, an *event sequence* refers to a sequence of executed nodes. More formally, an *event sequence* E is a sequence $<e_1,e_2, ..., e_k>$ of events where each *event* is a tuple e_i = (n_i, S_i) where $n_i \in N$ is a node and S_i is a set of variables referred to as a *constraint set*. For every two adjacent events, $e_i = (n_i,S_i)$ and $e_{i+1} = (n_{i+1},S_{i+1})$ there exists a definition clear path with respect to S_i from n_i to n_{i+1}.

The event sequence identifies a sequence of nodes that are to be executed during program execution. A constraint set associated with each event $e_i = (n_i,S_i)$ identifies the constraints imposed on the execution from a given node n_i to node n_{i+1} of the next event e_{i+1} in the event sequence. It is required that all variables in the constraint set are not modified during program execution between node n_i and node n_{i+1}. For example, the following is an event sequence E = <(s,∅), (10,{*max*}), (*n7*,{*max*}), (*n4*,∅), (*n5*,∅)> in the program of Figure 2, where *s* is the start point of the program. This event sequence requires that the start node s (start node of the main module) is first executed, followed by the execution of node 10, followed by the execution of node *n7*, followed by execution of node *n4* in function *A1*, and finally execution of node *n5*. A constraint is imposed on the execution between nodes 10 and *n7*; it is required that the value of *max* is not modified during program execution between nodes 10 and *n7* (similarly, between *n7* and *n4*). However, there are no constraints imposed on the execution between, for example, nodes s and 10, i.e., the execution can follow any path between nodes s and 10.

4.1. Generating Event Sequences

Initially, for the given goal node *g*, the following initial event sequence is generated: E_0=<(s,∅),(g,∅)>. Suppose that during the search process a problem node p is encountered, i.e., a test node at which the execution should be changed but the searching procedure is not able to find a program input that will alter execution at p. In this case, the chaining approach finds a set of last definitions of problem node p: d_1, d_2,..., d_N. This set is used to generate N event sequences. Each newly generated event sequence contains an event associated with problem node p and an event associated with last definition d_i. The following event sequences may be generated:

$E_1 = <(s,\emptyset), (d_1,D(d_1)), (p,\emptyset), (g,\emptyset)>$

...

$E_N = <(s,\emptyset), (d_N, D(d_N)), (p,\emptyset), (g,\emptyset)>$,

where $D(d_i)$ is a variable whose value is defined at d_i.

The chaining approach selects one of the event sequences from the list, for example E_1, and tries to find program input on which event sequence E_1 is traversed. If during this process a new problem node p_1 is encountered, e.g., between the start node s and d_1, the chaining approach determines a set of last definitions for p_1, $\{f_1, f_2,..., f_M\}$, and generates M new event sequences: $E1_1, E1_2,..., E1_M$. These event sequences are constructed from E_1 by inserting two events to E_1: one related to p_1 and the second related to f_i. The following event sequences may be generated:

$E1_1=<(s,\emptyset),(f_1,D(f_1)),(p_1,\emptyset),(d_1,D(d_1)),(p,\emptyset),(g,\emptyset)>$

...

$E1_M=<(s,\emptyset),(f_M,D(f_M)),(p_1,\emptyset),(d_1,D(d_1)),(p,\emptyset),(g,\emptyset)>$.

The chaining approach selects the next event sequence from the set of already generated sequences and tries to find program input on which this event sequence is traversed. If such an input is found then the input to execute the goal is also found. If the input is not found then new event sequence may be generated and the chaining approach selects the next event sequence to explore. This traversal process continues until the solution is found or the designated search resources are exhausted, e.g., time limit.

4.2. The Search Process

In this section we describe the process of finding input data to "traverse" a given event sequence. The problem is stated as follows: Given an event sequence $E = <e_1, e_2,..., e_k>$. The goal is to find program input **x** on which event sequence E is traversed. The search starts by initially executing a program for an arbitrary program input. During the program execution for each executed branch, the search procedure decides whether the execution should continue thorough this branch or an alternative branch should be taken. In the latter case a new input must be found to change the flow of execution at the current branch. For his purpose every branch in the program is classified into different categories. The branch classification is determined prior to the program execution. The search process uses this classification during program execution to continue or suspend the execution at the current branch. In the latter case, the search procedure determines a new program input.

For every two adjacent events $e_i=(n_i,S_i)$ and $e_{i+1}=(n_{i+1},S_{i+1})$ in the event sequence E a branch classification is computed. The branch classification relates to the situation that event e_i has already occurred (node n_i was executed) and now the goal is to reach node n_{i+1} without modifying any variable in S_i. A branch (p,q) is called a *critical* branch with respect to events e_i and e_{i+1} iff (*i*) there does not exist a definition clear path from p to n_{i+1} or n_i with respect to S_i through branch (p,q), and (*ii*) there exists a definition clear path from p to n_{i+1} with respect to S_i through the alternative branch of (p,q). If a critical branch (p,q) is executed, then program execution is suspended because the execution cannot lead to n_{i+1} or the path is not a definition clear path from q to n_{i+1} or n_i with respect to S_i. The search algorithm attempts to find a program input **x** which will change the flow of execution at critical branch (p,q); as a result, an alternative branch of (p,q) is executed. If a program input is found, the execution continues through the alternative branch. However, if program input **x** cannot be found, then the search is suspended and node p is "declared" as a problem node (for which new event sequences may be generated).

4.3. Finding input data

The problem of finding input data is reduced to a sequence of subgoals where each subgoal is solved using function minimization search techniques that use branch predicates to guide the search process. Each branch (p,q) in the flow graph is labeled by a predicate, referred to as a *branch predicate*, describing the conditions under which the branch will be traversed. There are two types of branch predicates: Boolean and arithmetic predicates. Boolean predicates involve Boolean variables. On the other hand, arithmetic predicates are of the following form: A_1 *op* A_2, where A_1 and A_2 are arithmetic expressions, and *op* is one of $\{<, \leq, >, \geq, =, \neq\}$. Each branch predicate A_1 *op* A_2 can be transformed to the equivalent predicate of the form: F *rel* 0. F is a real-valued function, referred to as a *branch function*, which is (1) positive (or zero if rel is <) when a branch predicate is false or (2) negative (or zero if rel is = or ≤) when the branch predicate is true. It is obvious that F is actually a function of program input **x** (more detailed discussion about the construction of a branch function can be found in [12]). The branch function is evaluated for any program input by executing the program.

Let $E = <e_1, e_2,..., e_m>$ be an event sequence. The goal is to find program input $\mathbf{x}$ on which E will be traversed. The goal of finding a program input $\mathbf{x}$ is achieved by solving a sequence of subgoals. Let $\mathbf{x}^0$ be the initial program input (selected randomly) on which the program is executed. If E is traversed, $\mathbf{x}^0$ is the solution to the test data generation problem. Suppose, however, that a critical branch (p,q) was executed between events e_i and e_{i+1}, i.e., the event sequence was partially traversed up to event e_i and the critical branch was encountered when the execution was supposed to reach node n_{i+1}. Let $E' = <e_1, e_2,..., e_i>$ be an event subsequence of E that was successfully traversed. In this case we have to solve the first subgoal. Let (p,t) be an alternative branch of branch (p,q). There are two possible cases depending on the type of branch predicate of (p,t). If a branch predicate of (p,t) is of a Boolean type then the search is terminated and node p is declared as a problem node. On the other hand, if branch predicate of (p,t) is of a arithmetic type then the following procedure is used: Let $F_1(\mathbf{x})$ be a branch function of branch (p,t). The first subgoal, is to find a value of $\mathbf{x}$ which causes $F_1(\mathbf{x})$ to be negative (or zero) at node p; as a result, (p,t) will be successfully executed. This problem is similar to the minimization problem with constraints because the function $F_i(\mathbf{x})$ can be minimized using numerical techniques for constrained minimization, stopping when $F_i(\mathbf{x})$ becomes negative (or zero, depending on rel_i). Because of a lack of assumptions about the branch function and constraints, direct-search methods [6] are used. The direct-search method progresses towards the minimum using a strategy based on the comparison of branch function values only. The simplest strategy of this form is that known as the *alternating variable* method, which consists of minimizing the current branch function with respect to each input variable in turn. The search proceeds in this manner until all input variables are explored in turn. After completing such a cycle, the procedure continuously cycles around the input variables until the solution is found or no progress (decrement of the branch function) can be made for any input variable. In the latter case, the search process fails to find the solution. Once input $\mathbf{x}^1$ for the first subgoal is found, the program continues execution until the whole event sequence is traversed or a new branch violation occurs at some other node. In the latter case, the second subgoal has to be solved. The process of solving subgoals is repeated until the solution $\mathbf{x}$ to the main goal node is found, or no progress can be made at some node, in which case, the search process fails to find the solution for a given event sequence; in the latter case, this node is declared as a problem node and new event sequences may be generated.

5. Experiment

The purpose of the experiment is to compare the assertion-oriented test generation with existing methods of test generation. This study has shown that some faults would go undetected if assertion-oriented test generation were not used. With respect to their ability to violate an assertion, we compare the proposed assertion-oriented test data generation to black box testing (represented by equivalence partitioning and boundary value analysis) and white box testing (represented by branch coverage). The motive of this comparison is not to suggest that our approach is better than or should replace these methods, but rather to see it as an improvement step toward revealing more software faults.

To derive this experiment, a set of 25 programs written in a subset of Pascal was used. We started by a set of nine original programs each of which contains one or more assertions. While keeping assertions untouched, from each original program we created a number of incorrect versions through error seeding where we introduce a fault in each version.

A brief description of the programs used in this experiment is now given. Program Bank performs simple banking operations such as opening an account, depositing and withdrawing money, and obtaining the balance. Program Minmax finds the minimum and the maximum in an array of integers. Program Total sums the elements of an array of integers. Program Average computes the average of an array of integers. Program RealNumberFormat converts a real number into a certain string format. Program Concatenation concatenates two strings. Programs RestrictedAverage and FunnyAverage compute the reciprocal average of an array elements, each in a different way. RestrictedSubstitute works on three strings. It substitutes each occurrence of the second string in the first string by the third string. The source code for these programs and their incorrect versions can be found in [16].

The experiment was performed on a Personal Computer with a 60mhz Pentium processor using the test data generation system TESTGEN [11, 15]. The measurement criterion of this experiment is the ability of a test data generation method to generate an input data that will reveal the fault. Specifically the experiment was conducted as follows. Each program was tested using black box, branch coverage, and assertion-oriented test data generation methods. The time limit of three minutes was imposed in the assertion test data generation to find program input on which an assertion was violated. If a test generation method is able to violate *at least* one assertion in an incorrect version of the program (or produce an incorrect output in the case of black-box and white-box testing) then this is considered as a success for this method. The complete result of the experiment is presented in Table I and should be interpreted as follows. Column 1 and Column 2 give the original program name and the number of incorrect versions (NV) created from this program, respectively. Column 3 gives the total number of assertions in the original program (NA). Columns 4, 5, and 6 give the number of versions of a program for which black box testing (BB), branch coverage testing (BC), and assertion-oriented testing (AT) were able to reveal a fault, respectively. For example, the second entry of Table I specifies that five versions were created from program Bank, and that this program has a total of three assertions. Out of five versions created from the Bank program, black box testing was able to violate at least one assertion in two versions (or produce an incorrect output), branch coverage testing did not violate any assertions in any of them (neither produce an incorrect output), and assertion-oriented testing violated one or more assertions in all five versions.

Table I. Experimental Results
NV: Number of Versions
NA: Number of Assertions
BB: Black-Box testing
BC: Branch Coverage
AT: Assertion-oriented Testing

Program Name	NV	NA	BB	BC	AT
Minmax	6	4	3	0	6
Bank	5	3	2	0	5
Total	1	1	0	1	1
Average	1	1	0	1	1
RealNumberFormat	1	1	0	0	1
Concatenation	2	2	2	0	2
RestrictedAverage	3	1	0	0	3
FunnyAverage	3	1	1	0	3
RestrictedSubstitute	3	3	1	0	1

Total	25	9	2	23
Percentage		36%	8%	92%

As shown in Table I, in a total of 25 incorrect programs considered in this experiment branch testing was successful 8% of the time, black box testing (represented by equivalence partitioning and boundary value analysis) 36% of the time, and assertion-oriented 92% of the time in generating an input data to reveal faults in incorrect programs. As stated earlier, the results of this initial experiment should not be interpreted as if we are suggesting that the assertion-oriented method is better than black box and white box testing methods. Our goal is to show that by using assertions, we can enhance the chances of detecting some faults that would be hard to detect otherwise.

As can be seen in Table I, assertion-oriented test data generation was able to violate one or more assertions in all of the programs considered. Although we are still in our initial experimentation with this new approach, the results are quite encouraging. Our next step is to conduct more experiments to assess the true value of this approach and to enhance it. Finally, the good error detection of the assertion-oriented test data generation as compared to other testing methods may be attributed to the way this method approaches the test data generation problem. It tries to find a test case that uncovers a fault as opposed to the existing testing methods that generate test cases to satisfy a selected testing criterion.

6. Conclusions

In this paper we have presented an approach of automated test data generation in which assertions are used to generate test cases. The goal of assertion-oriented testing is to identify program test on which an assertion(s) is violated. The assertion-oriented test generation is a very attractive approach of test data generation because each generated test case uncovers an error. We have shown that the existing methods of automated test data generation for white-box testing may be used to generate test cases that violate assertions. The assertion-oriented test data generation has been developed as a part of the test data generation system TESTGEN [11, 15]. Our initial experience with the assertion-oriented test data generation is very encouraging; this approach was able to find efficiently many faults that were not detected by others testing methods. However, more research is needed to better understand the advantages and limitations of this method of test data generation.

The effectiveness of the assertion-oriented test generation depends on the assertions provided by programmers (certain classes of assertions may be generated automatically from the source code). Assertions are mainly used in debugging, but we have shown that they can be used in the test data generation. Since the approach presented in this paper is automatic, programmers may be encouraged to provide assertions for the purpose of testing rather than debugging. In the light of this initial experiment and the promising performance of the assertion-oriented test data generation method, we think that directing testing resources toward assertion violation is a good investment in the tester's effort and resources, especially when this process is done automatically without human intervention.

We believe that the assertion-oriented test generation may encourage research in the area of developing more powerful methods of finding test cases on which assertions are violated. In addition, it may encourage research in assertion derivation by using formal methods. Some classes of assertions may be derived from program specification, e.g., precondition and post-conditions. Such assertions may be very powerful if used in the process of test generation.

Acknowledgments:
We would like to thank Timothy Thomas (a Master student) for his participation in the initial stages of the experiment.

7. References

[1] D. Bird, C. Munoz, "Automatic generation of random self-checking test cases," IBM Systems Journal, vol. 22, No. 3, 1982, pp. 229-245.

[2] B. Boehm, R. McClean, D. Urfig, "Some experience with automated aids to the design of large-scale reliable software,“ Proceedings of the International Conference on Reliable Software, 1975, pp. 105-113.

[3] 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.

[4] 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.

[5] R. DeMillo, A. Offutt, "Constraint-based automatic test data generation," IEEE Transactions on Software Engineering, vol. 17, No. 9, 1991, pp. 900-910.

[6] P. Gill, W. Murray, Ed., Numerical Methods for Constrained Optimization, New York: Academic, 1974.

[7] R. Ferguson, B. Korel, "The chaining approach for software test data generation," ACM Transactions on Software Engineering and Methodology (to appear).

[8] R. Holt, J. Cordy, "The turing programming language," Communications of ACM, vol. 31, No. 12, 1988, pp. 1410-1423.

[9] W. E. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," IEEE Transactions on Software Engineering, vol. 4, No. 4, 1977, pp. 266-278.

[10] C. Hulten, "Simple dynamic assertions for interactive program validation," AFIPS Conference Proceedings, Las Vegas, 1984, pp. 405-410.

[11] B. Korel, "TESTGEN - A structural test data generation system," Proceedings of the 6th International Conference on Software Testing, Washington, D.C., 1989.

[12] B. Korel, "Automated test data generation," IEEE Transactions on Software Engineering, vol. 16, No. 8, 1990, pp. 870-879.

[13] B. Korel, "A dynamic approach of automated test data generation," Conference on Software Maintenance, San Diego, 1990, pp. 311-317.

[14] B. Korel, "Dynamic method for software test data generation," Journal of Software Testing, Verification, and Reliability, vol. 2, 1992, pp. 203-213.

[15] B. Korel, "TESTGEN - An execution-oriented test data generation system," Technical Report TR-SE-95-01, Department of Computer Science, Illinois Institute of Technology, 1995.

[16] B. Korel, A. Al-Yami "Assertion-oriented test data generation," Technical Report TR-SE-95-04, Department of Computer Science, Illinois Institute of Technology, 1995.

[17] N. Levenson, S. Cha, J. Knight, T. Shimeall, "The use of self checks and voting in software error detection: An empirical study," IEEE Transactions on Software Engineering, vol. SE-16, No. 4, 1990, pp. 432-443.

[18] B. Meyer, Object-Oriented Software Construction. Prentice-Hall, 1988.

[19] 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.

[20] D. Rosenblum, "Toward a method of programming with assertions," Proceedings of the International Conference on Software Engineering, 1992, pp. 92-104.

[21] L. Stucki,, G. Foshee, "New assertion concepts for self-metric software validation," Proceedings of the International Conference on Reliable Software, 1975, pp. 59-71.

[22] J. Voas, K. Miller, "Putting assertions in their place," Proceedings of the International Symposium on Software Reliability Engineering, 1994.

[23] S. Yau, R. Cheung, "Design of self-checking software," Proceedings of the International Conference on Reliable Software, 1975, pp. 450-457.