Intelligent Assertions Placement Scheme for String Search Algorithms

Ali M. Alakeel
College of Computers and Information Technology
University of Tabuk
Tabuk, Saudi Arabia
alakeel@ut.edu.sa

***Abstract*— String search algorithms are found within the internal structure of most Information Retrieval Systems in military applications, aircraft software, medical applications, and commercial applications. Like any software, different errors may occur during the implementation of string search algorithms. Because of the wide range of applications that use string search algorithms, the consequences of these programming errors may be disastrous or fatal. This paper presents an intelligent assertions placement scheme for string search algorithms with the objective to enhance the testability of these algorithms during their testing phase. Instead of placing assertions randomly or after each statement of the program, our proposed method inserts assertions intelligently in a set of selected locations of the string search algorithm that are considered to be error prone or essential to the correctness of the program. The results of a small case study show that applying the proposed method may significantly increases the chances of detaching programming errors associated with string search algorithms that may go undetected using only traditional black-box and white-box testing methods.**

Keywords-assertions placement; string search algorithms; assertion-based software testing; software testing

I. INTRODUCTION

String search algorithms are found within the internal structure of most Information Retrieval Systems in military applications, aircraft software, medical applications, and commercial applications. The main function of a string search algorithm is to identify all instances of a give pattern *p* with size *m* characters that may exist in a text *t* with size *n* characters, such that $m<<n$ [1]. Like any software, different errors may occur during the implementation of string search algorithms. Because of the wide range of applications that use string search algorithms, the consequences of these programming errors may be disastrous or fatal. For example, if a medical information retrieval system fails to return the exact prescribed medicine, this action may jeopardize the patient's life. Also, if a military missile control system fails to retrieve the target's exact coordinates, the results could be disastrous [2]. Therefore, the correctness of the implementation of any string search algorithm is crucial.

Software correctness may be improved by applying thorough and rigorous software testing methods [16]. Program assertions are recognized as a supporting aid in detecting faults during software testing, debugging and maintenance [17-22]. Also, assertions have been shown to increase software testability [23-25]. Therefore, assertions may be inserted into software code in those positions that are considered to be error prone or have the potential to lead to software crash or failure.

Most string search algorithms, e.g., [3-15], share a common programming internal structure which may make them susceptible to the same type of errors during their implementations. For example, most string search algorithms are composed of two main parts: checking and skipping. These two major parts are considered the heart of any string matching algorithm where they involve the dealings and manipulations of certain elements. Some of these elements are the starting point of checking, the direction of checking, the skipping strategy, the number of static or dynamic reference characters, and different shift distances. For example, a common programming error that may occur during the implementation of a sting search algorithm it that the shift distance might become zero. Also, it is possible that the number of occurrences of the pattern *p* in text *t* found by the algorithm might be less than or greater than the actual occurrences of *p* in *t*. Therefore, the placement of programming assertions in the *proper* locations within string search algorithms may enhance the testability of these programs and leads to the detection of programming faults during the testing stage. It should be noted that the use of assertions for testing purpose should only be used as a complementary and an extra step after traditional testing methods such as black-box and white-box testing methods [3] have been applied to the software.

This paper presents an intelligent assertions placement scheme for string search algorithms with the objective to enhance the testability of these algorithms during their testing phase. Instead of placing assertions randomly or after each statement of the program, our proposed method inserts assertions intelligently in a set of selected locations of the string search algorithm that are considered to be error prone or essential to the correctness of the program. The results of a small case study show that applying the proposed method may increases the chances of detaching programming errors associated with string search algorithms that may go undetected using only traditional black-box and white-box testing methods.

The rest of this paper is organized as follows. Sections II and III provide a brief introduction to assertions-based

 ISBN: 978-1-61208-269-1

software testing and string search algorithms, respectively. Our proposed method for assertions placement in string search algorithms is presented in Section IV. In Section V, a case study is presented. Conclusions are presented in Section VI.

II. ASSERTION-BASED SOFTWARE TESTING

Software testing is a very labor intensive task and cannot by any means guarantees the correctness of any software or that the software is error-free. However, rigorous software testing may increase the confidence in the software under test. There are two main approaches to software testing: Black-box and White-box [3]. Test data generation is the process of finding program input data that satisfies a given criteria, e.g., [30, 31]. 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. Programmers usually start by testing their software using black-box methods against a given specification. By their nature black-box testing methods might not lead to the execution of all parts of the code. Therefore, this method may not uncover all faults in the program. To increase the possibility of uncovering program faults, white-box testing is then used to ensure that an acceptable coverage has been reached, e.g., branch coverage.

Program assertions are recognized as a supporting aid in revealing faults during software testing, debugging and maintenance [17-22]. Also, assertions have been shown to increase software testability [23-25] An Assertion specifies a constraint that applies to some state of computation. The state of an assertion is represented by two possible values: *true* or *false*. For example, *assert*(0<index<=100), is an assertion that constraints the values of some variable "index" to be in the range of 1 and 100 inclusive. As long as the values of "index" is within the allowed range the state of this assertion is *true*. Any other values beyond this range, however, will cause the state of this assertion to become *false* which indicates the violation of this assertion. Many programming languages support assertions by default, e.g., Java and Perl. For languages without built-in support, assertions can be added in the form of annotated statements. For example, [18] presents assertions as commented statements that are pre-processed and converted into Pascal code before compilation. Many types of assertions can easily be generated automatically such as boundary checks, division by zero, null pointers, variable overflow/underflow, etc. Beyond simple assertions that can easily be generated automatically, a method to generate more complex assertions for Pascal programs is presented in [18]. For this reason and to enhance their confidence in their software, programmers may be encouraged to write more programs with assertions. It should be noted, however, that writing the proper type of assertions and choosing the proper locations to inject them into the software is very important to the effectiveness of this methodology. Inserting assertions after every statement of the program is an extreme case scenario which can makes the whole process of assertions processing very costly and prohibitive [28]. Therefore is its imperative to devise a scheme for assertion's placement within the software under test such that assertions are only inserted in *selected* location within the program's code. Assertion-Based software testing [18, 19, 21], has been shown to be effective in detecting program faults as compared to traditional black-box and white-box software testing methods. Given an assertion *A*, the goal of Assertion-Based testing is to identify program input for which *A* will be violated. The main aim of Assertion-Based Testing is to increase the developer confidence in the software under test. Assertion-Based Testing is intended to be used as an extra and complimentary step *after* all traditional testing methods have been performed to the software. Assertion-Based Testing gives the tester the chance to think deeply about the software under test and to locate positions in the software that are very important with regard to the functionality of the software. After locating those important locations, assertions are added to guard against possible errors with regard to the functionality performed in these locations.

The process of writhing program assertions may depend heavily on the tester's experience and knowledge of the program under test. To aid in this process a simple tool may be used to automatically generate assertions in certain locations of the program, which guard against errors, such as division by zero, array boundary violations, uninitialized variables, stack overflow, null pointer assignment, pointer out of range, out of memory (heap overflow), and integer / float underflow and overflow [18]. However, there are application-specific locations in the program itself that may need to be guarded by assertions depending on the importance of these locations to the correctness of the application. For example, in string search algorithms, computing the location of the pattern in the input string and index manipulation during the checking and skipping process are very important to the correctness of these algorithms.

III. STRING SEARCH ALGORITHMS

The problem of string searching may be stated as follows. Given a text string *t* of size *n* and a pattern string *p* of size *m* (where $n >> m$), find all occurrences of *p* in *t* [1]. During our investigation of string matching algorithms, we noticed that most of the proposed algorithms are usually compared against classical exact string searching algorithm such as Naïve (brute force) algorithm and Boyer-Moore-Horsepool (BMH) algorithm [5]. Some of these algorithms preprocess both the text and the pattern, e.g., [3], while others need only to preprocess the pattern, e.g., [4, 5]. In all cases, the exact string searching problem consists of two major steps: checking and skipping. The checking step itself consists of two main stages. In the first stage main objective is to search along the text for a reasonable candidate string, while the second stage goal is to perform a detailed comparison of the candidate string, found in the first stage, against the pattern to verify the potential match. Some characters of the candidate string must be selected carefully in order to avoid the problem of repeated examination of

 ISBN: 978-1-61208-269-1

each character of text when patterns are partially matched. Intuitively, the fewer the number of character comparisons in the checking step the better the algorithm is.

Different string search algorithms may differ in the way they implement the checking process, e.g., [4, 5]. After the checking step, the skipping step shifts the pattern to the right to determine the next position in the text where the substring text can possibly match with the pattern. The reference character is a character in the text chosen as the basis for the shift according to the shift table. Some string search algorithms may use one or two reference characters and the references might be static or dynamic [18, 19]. Additionally, some algorithms focus on the performance of the checking operation while others focus on the performance of the skipping operation [10]. The shift distance used may differ from one string search algorithm to another; it ranges from only one position in the Naïve algorithm, up to m positions in Boyer-Moore-Horspool algorithm [5], m+1 positions in Raita's algorithm [4], and up to 3m+1 positions in CSA algorithm [11]. The following text provides a detailed description of the CSA sting search algorithm as reported in [11].

A. Checking and Skipping Algorithm (CSA)

A string search algorithm is a succession of checking and skipping, where the aim of a good algorithm is to minimize the work done during each checking and to maximize the length distance during the skipping. Most of the string matching algorithms preprocess the pattern before the search phase to help the algorithm to maximize the length of the skips, the preprocessing phase in the CSA algorithm helps in both increases, the performance of the checking step by converting some of the character-comparison into character-access and maximizes the length of the skips. At each attempt during the checking steps the CSA compares the character at *last_mismatch* (the character that causes the mismatch in the previous checking step) with the corresponding character in *Text*. If they match, the comparison goes from right to left, including the compared character at *last_mismatch*. The idea here is that the mismatched character must be given a high priority in the next checking operation. After a number of checking steps, this leads to start the comparison at the least frequent character without counting the frequency of each character in the text.

For the skipping step, CSA has five reference characters, including three static references and two dynamic references. The *Text* pointer *TextIx* always points to the character, which is next to the character corresponding to the last character in *Pat* and the reference character *ref* always points to the character that corresponds to the last character in *Pat* i.e. *ref* = *TextIx* – 1. Now let *ref1* = *TextIx*, then the reference character *ref2* can be calculated from *ref* or *ref*1, where *ref*2 can be found as "*ref2* = *TextIx* + *m* - 1" or "*ref2* = *TextIx* + *m*" depending on the existence of *ref* or *ref1* in *Pat*, where *ref2* = *TextIx* + *m* - 1 during the checking step if the character at *ref* doesn't exist in *Pat*, or *ref2* = *TextIx* + *m* after the checking step if the character at *ref1* doesn't exist in *Pat*. In addition to that, CSA pre-processes the pattern to produce two different arrays, namely *skip* and *pos*. Each array has a length equals to the alphabet size. The *skip* array is used when the reference character *ref1* exists in *Pat*, it expresses how much the pattern is to be shifted forward after the checking step. While the *pos* array defines where each one of the different reference characters *ref1*, *ref2*, *ref_ref1*, *or ref_ref2* is located in *Pat*, if any one of them exists in *Pat*, where the two dynamic pointers *ref_ref1* and *ref_ref2* can be calculated from two static pointers *ref1* and *ref2*, respectively. The CSA algorithm is designed to scan the characters of both the text and the pattern from right to left.

```
1,2, 3      #include <iostream>; #include <iomanip>;  #include <cstring>
4,5, 6      using std::cout; using std::cin; #define ASIZE 256
7           void PreProcessPat(char *, int , int *, int *);
8           void CSA(char *, int , char *, int , int *, int *);
9           int main(){
10          char Text[] = "test This is a test for string test";
11,12, 13   char Pat[] = "test";  int PatLength = 4; int TextLength = 35;
14,15, 16   int pos[1000]; int skip[1000]; cout<<Text;
17,18, 19   getchar();  cout<<Pat;  getchar();
20          PreProcessPat(Pat, PatLength, pos, skip);
21          CSA(Pat, PatLength, Text, TextLength, pos, skip);
22, 23, 24  cout<<Text;  getchar();  return 0; }
25  void PreProcessPat(char *Pat, int PatLength, int *pos, int *skip){
26          char c;
27          for(int j = 0; j<ASIZE; j++) {
28, 29        pos[j]=0; skip[j] = 2*PatLength;  }
30          for(int j=0; j<PatLength; j++) {
31, 32, 33    c = Pat[j]; pos[c]= j +1; skip[c] = 2 * PatLength - j -1; }}
34  void CSA(char *Pat, int PatLength, char *Text, int TextLength, int
*pos, int *skip){
35          int TextIx, PatIx, last_mismatch, z;
36          int pt, pt1, ref, ref1, ref_ref1, ref2, ref_ref2;
37          int infix[ASIZE] = {0};
38, 39, 40  infix[Pat[0]] = 1; last_mismatch =0; TextIx = PatLength;
41          while(TextIx<=TextLength+1) {
42  if(Text[TextIx - PatLength + last_mismatch] ==  Pat[last_mismatch])
43           if(infix[Text[TextIx - PatLength]]) {
44          for( z = 0, PatIx = PatLength - 1; PatIx; PatIx-- )
45          if(Text[TextIx - ++z] != Pat[PatIx]){
46             last_mismatch = PatIx;
47             goto next;  }
48  cout<<"\nAn occurrence at location  "<<TextIx-PatLength <<" to
"<<TextIx - 1<<"\n";     }
49          next:
50, 51      ref = TextIx - 1; ref1 = TextIx;
52          if ( !pos[Text[ref]] ){
53, 54, 55    ref2 = ref + PatLength; pt1 = pos[Text[ref2]]; ref_ref2 = ref2
+ PatLength - pt1;
56  TextIx += 3 * PatLength - pt1 - pos[Text[ref_ref2]];
} else {
57           pt = pos[Text[ref1]];
58          if( !pt){
59, 60, 61  ref2 = TextIx + PatLength; pt1 = pos[Text[ref2]]; ref_ref2 =
ref2 + PatLength - pt1;
62  TextIx += 3 * PatLength + 1 - pt1 - pos[Text[ref_ref2]]; } else {
63          ref_ref1 = ref1 + PatLength - pt;
64  TextIx += skip[Text[ref_ref1]] - pt + 1;
} }
}
65          return;
}
```

Figure 1. C++ implementation of CSA algorithm.

At each attempt, it first compares the character at *last_mismatch* with the corresponding character in *Text*; if they match, it compares the first character of *Pat* with the corresponding characters in *Text*, and if they match CSA compares the other characters from right to left including the character at *last_mismatch* and excluding the first character of *Pat*. Whether there is an occurrence of *Pat* in *Text* or not, the existence of the character at *ref* in *Pat* will be checked first, so there are two cases:

1) The character at *ref* exists in *Pat*: In this case, the existence of *Pat* in *Text* will be checked. After the checking step, the existence of *ref1* in *Pat* will be examined. Hence, there are two cases:

1.1) the character at *ref1* doesn't exist in *Pat*. Then *ref2* and *ref_ref2* will be calculated. Next, the pointer *TextIx* will be moved forward to align with the character at *ref_ref2*.

1.2) the character at *ref1* exists in *Pat*. Then *ref_ref1* will be calculated. Afterwards, the pointer *TextIx* will be moved forward to align with the character at *ref_ref1*.

2) The character at *ref* doesn't exist in *Pat*: In this case, *ref2* and *ref_ref2* will be calculated according to the pointer *ref*, then the pointer *TextIx* will be moved forward to align with the character at *ref_ref2*. Fig. 1 shows a C++ implementation of the CSA algorithm.

IV. INTELLIGENT ASSERTIONS PLACEMENT SCHEME FOR STRING SEARCH ALGORITHMS

In this section, we will describe in more details our proposed approach for intelligent assertions placement in string search algorithms.

A. Related Work

Assertions placement methods reported previously in the literature, e.g., [24, 26, 27], are mostly dependent on the intervention of the programmer and involves the analysis of all of the programs' code. In [24], a software tool is developed which assist the programmer in inserting assertions in a previously selected locations of C programs. There is no real placement strategy proposed by [24] other than what is proposed manually by the programmer. The usefulness of this tool is in converting assertions specified in pseudo-code into real programming code. Also, an assertion placement scheme designed specifically for embedded systems is proposed in [27]. Given a program P with a set of statements *S*, a heuristic presented in [26] that is based on propagation analysis [28] of each statement $s_k \in S$ found in the program, estimates the probability that a program fault at any statement, $s_k \in S$, will propagate to affect *negatively* the output of the program P. Based on this probability, this scheme selects those statements of the program that should be guarded by assertions. Although this heuristic is simple it is impractical for large commercial programs because most of its steps require human intervention. Additionally, this heuristic may ends with placing assertions after every statement of the program which makes it very expensive in terms of additional execution time. Our proposed method presented in this paper, only places assertions to guard those parts of the program that are considered to be *vital* to its functionality, therefore, minimizing the overhead that may be introduced by assertions processing.

B. Motivation

During our investigation of a set of string matching algorithms reported in the literature, e.g., [3-15], we found out that most of these algorithms share a common programming structure which makes them are susceptible to the same types of programming errors that may occur during their implementations. For example, it can be noticed that there are different factors and elements of string matching algorithms that may lead to program errors during the implementations of these algorithms into real program's code. Some of these elements are the starting point of checking, the direction of checking, the skipping strategy, the number of static or dynamic reference characters, and different shift distances. Therefore, like any software, it is possible that program errors may occur during the implementation of any string matching algorithm. For instance, the shift distance might become zero or the number of occurrences of the pattern *p* in text *t* found by the algorithm might be less than or greater than the actual occurrences of *p* in *t*. Moreover, it has been notice during this study that these types of errors are not easily detected by traditional black-box and white box software testing methods [16].

Based on the properties of the internal structure of string search algorithms, this paper proposes an assertions placement strategy that *intelligently* guides the programmer to the locations in which assertions should be placed. As will be described shortly, our proposed method employs data dependency analysis [29] on those parts of the program that are *vital* to its functionality. Through our investigation of string matching algorithms, the checking and skipping components are the most important parts. Data dependency analysis is described as follows. Given a program P with as set of statements, $S = \{s_1, s_2, s_3, \ldots, s_n\}$ and a set of variables, $V = \{v_1, v_2, v_3, \ldots, v_m\}$, form which any statement, $s_k \in S$, may be composed, data dependency analysis defines the relationships between the elements of the set of program statements *S* with respect to the usage and modifications of the set of variables *V*. Formally, there exists a data dependency between two statements *s*i and *s*j such that $j > i$ in their order of appearance in the program, with respect to a variable, $v_t \in V$, if the following three conditions holds. (1) The statement s_i assigns a value to *v*, and (2) the variable v_t is used at the statement s_j, and (3) there exists a program control path between s_i and s_j, in which the variable v_t is not modified. For example, in the program of Fig. 1, there exists a data dependency between nodes 57 and 63 because the variable "pt" is assigned a value at node 57, node 63 uses variable "pt", and there exists a program control path, from node 57 to node 63, in which "pt" is not modified. This program control path is: (57, 58, 63). The data dependencies in the program may be represented by data dependency graph [31], such that the nodes of the graph represent program statements and the directed arcs represent data dependencies.

 ISBN: 978-1-61208-269-1

```
1,2, 3      #include <iostream>; #include <iomanip>; #include <cstring>
4,5, 6      using std::cout; using std::cin; #define ASIZE 256
7           void PreProcessPat(char *, int , int *, int *);
8           void CSA(char *, int , char *, int , int *, int *);
9           int main(){
10          char Text[] = "test This is a test for string test";
11          char Pat[] = "test";
12,13, 14   int PatLength = 4; int TextLength = 35; int pos[1000];
15          int skip[1000];
16,17,18, 19 cout<<Text; getchar(); cout<<Pat;  getchar();
20          PreProcessPat(Pat, PatLength, pos, skip);
21          CSA(Pat, PatLength, Text, TextLength, pos, skip);
22, 23      cout<<Text;  getchar();
24          return 0; }
25  void PreProcessPat(char *Pat, int PatLength, int *pos, int *skip){
26          char c;
27          for(int j = 0; j<ASIZE; j++) {
/* A1: (j>=0)and(j<ASIZE) */        // Assertion No. 1
28, 29        pos[j]=0; skip[j] = 2*PatLength; }
30          for(int j=0; j<PatLength; j++) {
/* A2: (j>=0)and(j<PatLength) */        // Assertion No. 2
31, 32, 33    c = Pat[j]; pos[c]= j +1; skip[c] = 2 * PatLength - j -1; }}
34  void CSA(char *Pat, int PatLength, char *Text, int TextLength, int
*pos, int *skip){
35          int TextIx, PatIx, last_mismatch, z;
36          int pt, pt1, ref, ref1, ref_ref1, ref2, ref_ref2;
37          int infix[ASIZE] = {0};
/* A3: (Pat[0]>=0)and(Pat[0]<ASIZE)*/ // Assertion No. 3
38, 39, 40  infix[Pat[0]] = 1; last_mismatch =0; TextIx = PatLength;
41          while(TextIx<=TextLength+1) {
/* A4: ((TextIx - PatLength + last_mismatch)>=0)and((TextIx - PatLength +
last_mismatch)<TextLength) *   // Assertion No. 4
42          if(Text[TextIx - PatLength + last_mismatch] ==
            Pat[last_mismatch])
/* A5:  ((Text[TextIx - PatLength])>=0)and((Text[TextIx - PatLength])<ASIZE)
*/  // Assertion No. 5
43          if(infix[Text[TextIx - PatLength]]) {
// Check the occurrence of Pat in Text from right to left excluding first
character
44          for( z = 0, PatIx = PatLength - 1; PatIx; PatIx-- )
/* A6: ((TextIx - ++z)>=0)and((Text[TextIx - tLength])<TextLength)*/ //
Assertion No. 6
45          if(Text[TextIx - ++z] != Pat[PatIx]){
46. 47        last_mismatch = PatIx; goto next; }
48          cout<<"\nAn occurrence at location  "<<TextIx-PatLength <<"
to "<<TextIx - 1<<"\n"; }
49          next:
50, 51      ref = TextIx - 1; ref1 = TextIx;
/* A7:  ((Text[ref])>=0)and((Text[ref])<1000)*/ // Assertion No. 7
52    if ( !pos[Text[ref]] ){
/* A8: ((Text[ref2])>=0)and((Text[ref2])<1000)*/  // Assertion No. 8
53,54   ref2 = ref + PatLength; pt1 = pos[Text[ref2]];
55  ref_ref2 = ref2 + PatLength - pt1;
/* A9: ((Text[ref_ref2])>=0)and((Text[ref_ref2])<1000) */ // Assertion No. 9
56  TextIx += 3 * PatLength - pt1 - pos[Text[ref_ref2]];
} else {
/* A10: ((Text[ref1])>=0 and ((Text[ref1])<1000) */  // Assertion No. 10
57, 58      pt = pos[Text[ref1]];  if( !pt){
/* A11: ((Text[ref2])>=0) and ((Text[ref2])<1000) */  // Assertion No. 11
59, 60  ref2 = TextIx + PatLength; pt1 = pos[Text[ref2]];
61  ref_ref2 = ref2 + PatLength - pt1;
/* A12: ((Text[ref_ref2])>=0) and ((Text[ref_ref2])<1000) */  //Assertion No. 12
62  TextIx += 3 * PatLength + 1 - pt1 - pos[Text[ref_ref2]]; }
else {
63          ref_ref1 = ref1 + PatLength - pt;
/* A13: ((Text[ref_ref1])>=0) and ((Text[ref_ref1])<1000) */  // Assertion No. 13
64  TextIx += skip[Text[ref_ref1]] - pt + 1; }
} }
65          return; }
```

Figure 2. CSA string search algorithm with assertions.

C. Proposed Intelligent Assertions Placement Scheme

Given a program P that represents an implementation of a string matching algorithm, our proposed method for intelligent assertions placement in string matching algorithms, proceeds in three main stages as follows. In the first stage, the checking and skipping components are identified. Also, the boundaries statements of each part are marked. Note that this step is performed manually. Giving these marked points, the second stage performs, automatically, a data dependency analysis of every statement within the marked boundaries of the checking and skipping components. The outcome of the first stage is a set of data dependency sub-graphs of every statement in the checking and skipping parts of the program P. For every statement *s*k, that lies within the marked boundaries of the checking and skipping boundaries, each data dependency sub-graphs will be composed of the program statements and data dependencies of the program's data dependence graph for which there exists a path that leads to the statement, *s*k. Finally, data dependency sub-graphs are then used, in the third stage, to produce a road map that will guide the process of our assertions placement strategy. For example, after applying the proposed method on the CSA program shown previously in Fig. 1, thirteen assertions were placed in selected locations. Fig. 2 shows a new version of the CSA algorithm after assertions have been placed within its code according to the proposed method in this research.

V. A CASE STUDY

The goal of this small case study is to show that applying the proposed method for assertions placement may significantly enhances the testability of string search algorithms, therefore, increasing the chances of detecting programs errors that may exist in these programs. In this case study, the CSA string search algorithm [11], is implemented by three different programmers with 3-5 years of experience. This stage produced three different versions of the CSA algorithm. Each of these versions is subjected to traditional black-box and white-box software testing methods. Specifically, the following software testing methods were used: black-box testing as represented by boundary value analysis and equivalence class partitioning while white-box testing is represented by branch coverage. Errors detected during these tests were fixed and this process is repeated until these methods fail to uncover to detect any faults.

In order to increase our confidence in these programs, our proposed scheme for assertions placement, described previously in Section IV, is applied to the three versions of the CSA string search algorithm. For each version, the outcome of this stage is a modified copy with assertions placed at selected location recommend by the proposed method to be error prone or crucial to the correctness of the CSA algorithm. For example, in the version of the CSA algorithms shown in Fig. 1, thirteen assertions were inserted in this version as shown in Fig. 2.

 ISBN: 978-1-61208-269-1

TABLE I. RESULTS OF CASE STUDY

#Violations	#Assertions Inserted	Program Name
1	13	CSA version#1
2	15	CSA version#2
0	10	CSA version#3

Note that some assertions are inserted automatically without the intervention of the programmer, while more complex assertions are developed manually by the programmer and inserted in the recommended locations as proposed method. Assertions that are generated automatically are array boundary checks, division by zero, null pointers and variable overflow/underflow. In the final stage of this case study, Assertion-Based software testing [18] is performed on each version of the CSA with assertions. Assertion-Based software testing main objective is to generate program's input data for which a given assertion is violated. If this assertion is violated, then a program fault has been uncovered [18]. As stated in [18], Assertion-Based software testing is intended to be used as an extra and complimentary step *after* all traditional testing methods, such as black-box and white-box [16], have been performed on each original copy of each program used in this case study. The result of this case study is shown in Table I. It should be noted that the result of this experiment may be different for different programs with different types of assertions.

As reported in Table I, using our proposed method for assertions placement together with Assertion-Based software testing, we were able to uncover program faults in two out of the three versions of the CSA string search algorithm used in this case study. This is encouraging results considering that all of these faults were not detected by traditional black-box and white-box software testing methods during the first stage of this study. Also, notice that each assertion's violation means that at least *one* fault has been uncovered.

VI. CONCLUTIONS AND FUTURE WORK

This research proposed a new method for intelligent assertions placement in string search algorithms. The proposed method main objective is to increase the testability of string search algorithms and to enhance the delectability of program faults during their testing phase. The proposed method is intended to be used as a pre-step before Assertion-Based software testing is performed on string search algorithms. The result of a case study, conducted to evaluate the proposed method, shows that using this method may significantly enhances the chances of detecting program faults associated with string search algorithms that may go undetected by applying only traditional software testing methods. Our future research concentrates on conducting an experimental study to evaluate the proposed method in wider range of string search algorithms and to investigate the applicability of this method in other applications software.

REFERENCES

[1] G. Stephen, "String Searching Algorithms", World Scientific, Singapore, 1994.

[2] [http://www.pcworld.com/article/110035/software_bug_may_cause_missile_errors.html]. Retrieved: March 6, 2013.

[3] P. Fenwick, "Fast string matching for multiple searches", Software–Practice and Experience, Vol. 31, No. 9, pp. 815–833, 2001.

[4] T. Raita, "Tuning the Boyer-Moore-Horspool String Searching Algorithm", Software Practice and Experience, Vol. 22, No. 10, pp. 879-844, 1992.

[5] R.S. Boyer and J.S. Moore, "A fast string searching algorithm", Communications of the ACM, Vol. 20, No. 10, pp. 762–772, 1977.

[6] M. S. Ager, O. Danvy, and H. K. Rohde, "Fast partial evaluation of pattern matching in strings", ACM/SIGPLAN Workshop Partial Evaluation and Semantic-Based Program Manipulation, San Diego, California, USA, pp. 3 – 9, 2003.

[7] K. Fredriksson and S. Grabowski, "Practical and Optimal String Matching", Proceedings of SPIRE'2005, Lecture Notes in Computer Science 3772, pp. 374-385, Springer Verlag, 2005.

[8] P. Smith, "On Tuning the Boyer-Moore-Horspool String Searching Algorithm", Short Communication, Software Practice and Experience, Vol. 24, No. 4, pp. 435-436, 1994.

[9] M. Mhashi, "The Effect of Multiple Reference Characters on Detecting Matches in String Searching Algorithms," Software Practice and Experience, Vol. 35, No. 13, pp. 1299 -1315, 2005.

[10] Mhashi, M., "The Performance of the Character-Access On the Checking Phase in String Searching Algorithms", Transactions on Informatica, Systems Sciences and Engineering, Vol. 9, pp. 38 –43, 2005.

[11] M. Mhashi and M. Alwakeel, "New Enhanced Exact String Searching Algorithm" IJCSNS International Journal of Computer Science and Network Security, Vol. 10, No. 4, pp. 13 – 20, 2010.

[12] R. N. Horspool, "Practical fast searching in strings," Software - Practice & Experience, Vol. 10, No. 6, pp. 501-506, 1980.

[13] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms, "IBM J. Res. Dev., Vol. 31, No. 2, pp. 249-260, 1987.

[14] A. Apostolico and M. Crochemore, "Optimal canonization of all substrings of a string," Information and Computation, Vol. 95, No. 1, pp. 76-95, 1991.

[15] L. Colussi, "Correctness and efficiency of the pattern matching algorithms," Information and Computation, Vol. 95, No. 2, pp. 225-251, 1991.

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

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

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

[19] A. Alakeel, "An Algorithm for Efficient Assertions-Based test Data Generation," Journal of Software, vol. 5, No. 6, pp. 644-653, 2010.

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

[21] A. Alakeel, "A Framework for Concurrent Assertion-Based Automated Test Data Generation," European Journal of Scientific Research, Vol. 46, No. 3, pp. 352-362, 2010.

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

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

 ISBN: 978-1-61208-269-1

[24] H. Yin and J.M. Bieman, " Improving Software Testabiaity with Assertion Insertion," Proceedings of Inernational Test Conference, pp. 831-839 October 1994.

[25] T. Tsai, C. Huang C., and J. Chang, "A Study of Applying Extended PIE Technique to Sofware Testability Analysis," Proceedings of IEEE Inter. Computer Software and Application Conf., pp. 89-98, 2009.

[26] J. Voas, "Software Testability Measurment for Intelligent Assertion Placement," Software Quality Journal (6), pp. 327-335, 1997.

[27] V. Izosimov, et. al., "Optimization of Assertion Placment in Time-Constarined Embedded Systems," Proceedings of The Sixteenth IEEE European Test Symposium, pp. 171-176, 2011.

[28] J. Voas, "PIE: A Dynamic Failure-Based Technique," IEEE Trans. on Software Eng., Vol. 18, No. 8, pp. 717-727, August, 1992.

[29] B. Korel, et. al., "Data Dependence Based Testability Transformation in Automated Test Generation," Proceedings of The 16th IEEE Inter. Symposium on Software Reliability Engineering, pp. 245-254, 2005.

[30] P. McMinn, "Search-Based Software Test Data Generation: A Survey," Software Testing, Verification and Reliability, Vol. 14, pp. 105-156, 2004.

[31] M. Harman and P. McMinn, "A theoretical and empirical study of search based testing: Local, global and hybrid search," IEEE Transactions on Software Engineering, Vol. 36, No. 2, pp. 226-247, 2010.

 ISBN: 978-1-61208-269-1