Using Assertion-Based Testing in String Search Algorithms

Ali M. Alakeel[1] and Mahmoud M. Mhashi[2]
College of Computers and Information Technology
University of Tabuk
P.O.Box 1458, Tabuk 71431, Saudi Arabia
alakeel@ut.edu.sa[1]
mmhashi@ut.edu.sa[2]

***Abstract*—Software programs may contain faults that cause them to work improperly. Assertion-Based testing has been shown to be effective in detecting program faults as compared to traditional black-box and white-box software testing methods. String search algorithm problem is one of the most important problems that had been investigated by many studies to find all the occurrences of a pattern (with size m characters) occurs in text (with size n characters), where m<<n. String search algorithms are one of the main elements of Information Retrieval Systems which are found in a wide range of applications such as military applications, aircraft software, medical applications, and commercial applications. Therefore, the correctness of any string search algorithms is vital. Different errors might occur during the implementation of any of these algorithms. An example of error, if the shift distance becomes zero, then the algorithm will not move forward. In this research, we show that Assertion-Based software testing may be effective in uncovering software faults associated with string searching algorithms. Our preliminary experimentation with this approach shows that several types of errors associated with string searching algorithms are uncovered using Assertion-Based software testing.**

Keywords-Software testing; Assertion-Based Testing; Program Assertions; String Search Algorithms.

I. INTRODUCTION

Software programs may contain faults that cause them to work improperly. The effects of software failure could be very disastrous and life threatening. For example, a software failure may cause an airplane to crash, a nuclear factory to meltdown or even to cause a military missile to hit the wrong target, e.g., [22]. For this reason, software testing methods has gained so much attention from researchers and industry practitioners since computers were invented.

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, thorough and 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. Test data generation is the process of finding program input data that satisfies a given criteria. Test generators that support black-box testing create test cases by using a set of rules and procedures; the most popular methods include equivalence class partitioning, boundary value analysis, cause-effect graphing. 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.

Assertion-Based testing [4-5, 7] 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.

String search algorithms are one of the main elements of Information Retrieval Systems (IRS) which are found in a wide range of applications such as military applications, aircraft software, medical applications, and commercial applications. If an information retrieval system fails to return the correct piece of information, the results could be disastrous. 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 missile control system fails to retrieve the exact coordinates of the target, the results could be disastrous. Therefore, the correctness of any string search algorithms is vital. String searching algorithms are so fundamental that most computer programs use them in one form or another.

In this paper, we show that Assertion-Based software testing [4-5, 7] may be effective in uncovering software faults associated with string searching algorithms. Since the time Assertion-Based testing was proposed in [4] and to the best of our knowledge, this research is the first to present an application a proposed testing methodology, i.e., Assertion-Based testing, to a well known reported algorithms, i.e., string search algorithms. For the purpose of this research we have selected a number of well-known published string searching algorithms, gave them to programmers to implement them and then used Assertion-Based software testing to test these

 ISBN: 978-1-61208-168-7

implementations. Our result is presented in a case study presented in Section IV of this text. It should be noted that the efficiency, performance or the competency of each string search algorithm, considered in our study, are not questioned. Our main objective is to show that Assertion-Based testing may be effective during the development and testing of such algorithms.

The rest of this paper is organized as follows. Related work is discussed in Section II. Section III presents an application of Assertion-Based software testing method to the string search algorithms. In Section IV, we present a case study, followed by our conclusion and future work in Section V.

II. RELATED WORK

A. String Search Algorithms

Exact string searching is one of the most important problems that had been investigated by many studies, e.g., [8-21]. The problem of string searching may be stated as follows. Given a text string (Text) of size n and a pattern string (Pat) of size m (where n >> m), find all occurrences of *Pat* in *Text* [8].

As reported in the literature, many exact string searching and pattern matching algorithms were introduced and their performance was investigated against classical exact string searching algorithm such as Naïve (brute force) algorithm and Boyer-Moore-Horsepool (BMH) algorithm. Some of these algorithms preprocess both the text and the pattern, e.g., [9] while others need only to preprocess the pattern, e.g., [10, 11]. In all cases, the exact string searching problem consists of two major steps: checking and skipping. The checking step itself consists of two phases:

1) A search along the text for a reasonable candidate string

2) A detailed comparison of the candidate 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 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., [12, 13]. 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 [14, 15]. Additionally, some algorithms focus on the performance of the checking operation while others focus on the performance of the skipping operation [16]. 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 [11], m+1 positions in Raita's algorithm [10], and up to 3m+1 positions in CSA algorithm [17].

From the previous discussion, it can be noticed that there are different factors and elements of string search 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. Thus it is possible that errors might occur during the implementation of any string searching algorithm. For instance, the shift distance might become zero or the number of occurrences of *Pat* in *Text* found by the algorithm might be less than or greater than the actual occurrences of *Pat* in *Text*. In a case study, presented in Section IV, we found out that Assertion-Based testing may be effective in catching some of the faults associated with string search algorithms.

B. Assertion-Based Softwre Testing

Assertions are recognized as a supporting aid in revealing faults during software testing, debugging and maintenance, e.g., [1-7]. Therefore, many programmers place them within their code in positions considered error prone or have the potential to lead to software crash or failure. 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, [4] 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, reference [4] presents a method to generate more complex assertions for Pascal programs. 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 that writing the proper type of assertions and choosing the proper locations to inject them into programs depend heavily on the tester's experience and knowledge of the program under test. As mentioned previously, 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 [3]. 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 searching 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 such algorithms.

 ISBN: 978-1-61208-168-7

```
1   #include <iostream>
2   #include <iomanip>
3   #include <cstring>
4   using std::cout;
5   using std::cin;
6   #define ASIZE 300
7   #define XSIZE 300
8   void preBmBc(char *x, int m, int bmBc[]) {
9    int i;
10  for (i = 0; i < ASIZE; ++i)
11  bmBc[i] = m;
12  for (i = 0; i < m - 1; ++i){
/* A1: (x[i]>=0 and x[i]<ASIZE) */                    // Assertion No. 1
13  bmBc[x[i]] = m - i - 1;
    }
    }
14  void suffixes(char *x, int m, int *suff) {
15  int f, g, i;
16  suff[m - 1] = m;
17  g = m - 1;
18  for (i = m - 2; i >= 0; --i) {
/* A2: (i + m - 1 - f)>=0 and (i + m - 1 - f)<XSIZE) */ // Assertion No. 2
19   if (i > g && suff[i + m - 1 - f] < i - g){
/* A3: (i)>=0 and (i)<XSIZE) */                       // Assertion No. 3
20                  suff[i] = suff[i + m - 1 - f];
      }
21  else {
22  if (i < g)
23  g = i;
24  f = i;
25  while (g >= 0 && x[g] == x[g + m - 1 - f])
26          --g;
/* A4: (i)>=0 and (i)<XSIZE) */                       // Assertion No. 4
27  suff[i] = f - g;
      }
    }
    }
28  void preBmGs(char *x, int m, int bmGs[]) {
29  int i, j, suff[XSIZE];
30  suffixes(x, m, suff);
31  for (i = 0; i < m; ++i)
32  bmGs[i] = m;
33  j = 0;
34  for (i = m - 1; i >= 0; --i)
35  if (suff[i] == i + 1)
36  for (; j < m - 1 - i; ++j)
37  if (bmGs[j] == m)
38  bmGs[j] = m - 1 - i;
39  for (i = 0; i <= m - 2; ++i){
/* A5: (m - 1 - suff[i])>=0 and (i)<XSIZE) */         // Assertion No. 5
40  bmGs[m - 1 - suff[i]] = m - 1 - i;
    }
    }
41  void BM(char *x, int m, char *y, int n) {
42  int i, j, bmGs[XSIZE], bmBc[ASIZE];
     /* Preprocessing */
43  preBmGs(x, m, bmGs);
44  preBmBc(x, m, bmBc);
     /* Searching */
45  j = 0;
46  while (j <= n - m) {
47  for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
48  if (i < 0) {
49  cout<<"\nAn occurrence at location  "<<j;
50  j += bmGs[0];
      }
51  else{
/* A6: (i>=0 and i<XSIZE) */                          // Assertion No. 6
/* A7: ((y[i + j])>=0 and (y[i + j])<ASIZE) */        // Assertion No. 7
52  if(bmGs[i] > bmBc[y[i + j]] - m + 1 + i)
53          j += bmGs[i];
54          else
55          j += bmBc[y[i + j]] - m + 1 + i;
    }
    }
    }
56  int main()
    {
57  char Text[] = "test This is a test for string test";
58  char Pat[] = "test";
59  int m = 4;
60  int n = 35;
61  cout << "\nInput text: " << Text << "\nPattern: " << Pat;
62  BM(Pat, m, Text, n);
63  cout<< "\n Press ENTER to exit ....";
64  getchar();
65  return 0;
}
```

Figure 1. Boyer-Moore Algorithm with Assertions

During our case study presented in Section IV, assertions were injected in locations that are error-prone and crucial to the correctness of a string search algorithm. This knowledge is gained through our investigation of each string search algorithm considered during our experiment.

III. USING ASSERTION-BASED TESTING IN STRING SEARCH ALGORITHMS

In this section, we present an application of Assertion-Based software testing to the Boyer–Moore string search algorithm [11]. Our complete case study is presented in section IV.

A. Boyer–Moore Algorithm

The Boyer–Moore string search algorithm [11] is a particularly efficient string searching algorithm. It has been the standard benchmark for the practical string search literature. The algorithm preprocesses the target string (key) that is being searched *for*, but not the string being searched *in* (unlike some algorithms that preprocess the string to be searched and can then amortize the expense of the preprocessing by searching repeatedly). The execution time of the Boyer-Moore algorithm, while still linear in the size of the string being searched, can have a significantly lower constant factor than many other search algorithms: it doesn't need to check every character of the string to be searched, but rather skips over some of them. Generally the algorithm gets faster as the key being searched for becomes longer. Its efficiency derives from the fact that with each unsuccessful attempt to find a match between the search string and the text being searched, it uses the information gained from that attempt to rule out as many positions of the text as possible where the string cannot match. Figure 1 shows an implementation of Boyer-Moore Algorithm after assertions have been added to it. In this program we inserted a total of seven assertions in different positions of the code. Assertions are numbered and shown in bold in Figure 1.

After applying Assertion-Based Testing to Boyer-Moore Algorithm of Figure 1 Assertion #2 was *violated*. As described in [4], during Assertion-Based Testing, each assertion found in the program under test is converted into a corresponding code

during a pre-processing stage. For example, Assertion#2 of Figure 1 will be converted into the following code:

```
18.1      if (!((i + m - 1 - f)>=0))
18.1.1          cout << "\nAssertion 2a Violation!";
18.2      if (!((i + m - 1 - f)<XSIZE))
18.2.1          cout << "\nAssertion 2b Violation!";
```

Considering the above segment of code, according to Assertion-Based Testing presented in [4], Assertion#2 is violated if either of statements 18.1.1 or 18.2.1 is executed. During our experiment, Assertion #2 was violated through the execution of statement 18.1.1. The violation of Assertion #2 has detected a fault in this program which is caused by the use of an uninitialized variable "f" used in statement#19 in Figure 1. Note that uninitialized variables might cause very serious bugs in the program due to the nondeterministic values those variables might take during the course of different program executions. It should also be noted that many forms of uninitialized variable go undetected by C++ compiler.

IV. CASE STUDY

For the purpose of this case study, we have selected, from the literature, seven different string searching algorithms. These algorithms are implemented in C++ by three programmers with more than 5 years of experience in software development. C++ language was selected because it is widely used in the industry in our area and in America. The programs are executed and tested using traditional 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. Using these traditional software testing methods, we were not able to uncover any faults in any of the seven programs. As stated in [4], 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. Therefore, and in order to uncover any faults, we injected assertions in certain locations of each of the selected string search algorithms used in this study and then applied Assertion-Based Testing as described in [4]. As reported in Table I, we were able to uncover program faults, in all of the seven programs, which were left uncovered by traditional software testing methods or by tests performed by the original authors of those string matching algorithms.

Information presented in Table I may be interpreted as follows. Column#1 and Columun#2 show the name of the string search algorithm and the number of assertions inserted in each one, respectively. Column#3 shows the number of assertions that were violated during Assertion-Based software testing. For example, row#3 of Table I shows that in an implementation of the Horespool algorithm [18], a total of three assertions were inserted in this program. Two out the three assertions (66.7%) were violated during Assertion-Based testing. In this case study, the number of assertions ranges from 3 to 18 assertions. The percentage of assertion violations ranges from 5.5% to 66.7% and the percentage of the tested algorithms that contains faults was 100%. It should be noted that the result of this experiment might be different for different programs with different types of assertions. The purpose of this case study is only to show that Assertion-Based Testing [4-5, 7] may be effective in detecting program faults when applied to the considered set of string search algorithm implementations used in this experiment. We emphasize that the quality and the merits of the string search algorithms themselves are not questioned and is beyond the scope of this research.

TABLE I. CASAE STUDY RESULTS

Algorithm's Name	#Assertions	#Violations
Boyer-Moore Algorithm	7	1
CSA Algorithm	13	1
Horespool Algorithm	3	2
KR Algorithm	4	1
AXAMAC Algorithm	9	1
COLUSSI Algorithm	18	1

V. CONCLUSION and FUTURE WORK

In this paper, we presented a new approach in which Assertion-Based testing is utilized to find software faults associated with string searching algorithms. We performed a case study in which a set of well-known string search algorithms are implemented and tested. During this case study, assertions were inserted in selected locations of each subject program to guard against possible errors. The result of this case study is encouraging and shows that Assertion-Based software testing was able to uncover faults in these programs that were overlooked by traditional software testing methods such as black-box and white- box testing. This result indicates that Assertion-Based testing may be very effective during the development and testing of string search algorithm. For our future research, we intend to extend our case study to include a wider range of string search algorithms especially those which function as a part of bigger commercial applications.

REFERENCES

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

[2] Yau S. and Cheung R., "Design of Self-Checking Software," Proceedings of the International Conference on Reliable Software, pp. 450-457, 1975.

[3] Rosenblum, D., "Toward A Method of Programming WithAssertions," Proceedings of the International Conference on Software Engineering, pp. 92-104, 1992.

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

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

 ISBN: 978-1-61208-168-7

[6] Korel B. and Alyami A., “Automated regression test generation,” Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, pp. 143 – 152, 1998.

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

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

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

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

[11] Boyer RS. and Moore JS., "A fast string searching algorithm", Communications of the ACM, Vol. 20, No. 10, pp. 762–772, 1977.

[12] Ager M. S., Danvy O., and Rohde H. K., "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.

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

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

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

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

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

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

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

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

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

[22] [http://www.pcworld.com/article/110035/software_bug_may_cause_missile_errors.html]. Last access date: July 26, 2011.

 ISBN: 978-1-61208-168-7