A GA-Based Approach to Automatic Test Data Generation for ASP.NET Web Applications

Islam T. Elgendy, Moheb R. Girgis, and Adel A. Sewisy

***Abstract*—One of the major challenges in software testing is the generation of test data automatically that satisfy a specified adequacy criterion. This paper presents a GA-based approach and a supporting tool for data-flow test data generation for ASP.NET web applications. The proposed tool accepts as input the web application under test, instruments it, and performs static analysis to compute the definition-use pairs. The proposed GA conducts its search by constructing new test data from previously generated test data that are evaluated as effective test data. In this GA, the chromosome is a collection of user interface control objects, where each control is considered as a gene. Therefore, novel crossover and mutation operators are developed to manipulate the chromosome, which are called *block crossover* and *control-based mutation* operators. The proposed GA accepts as input the instrumented version, the list of definition-use pairs to be covered, and input controls related information. The tool produces a set of test cases, the set of definition-use pairs covered by each test case, and a list of uncovered definition-use pairs, if any. Also the paper presents a case study to illustrate how the tool works. Finally, it presents the results of the empirical evaluation that is performed to evaluate the effectiveness of the generated test data in exposing web application errors.**

***Index Terms*—Software testing, Data Flow Testing, Automatic test data generation, Automated Testing Tool, Web Applications Testing.**

I. Introduction

SOFTWARE testing has been widely used in the industry as a quality assurance technique for the various artifacts in a software project. However, software testing is very labor-intensive, time-consuming and expensive; almost 50% of the cost [1], [2] and 40% of the time of a software system development is spent on software testing [3]. Software testing has two main aspects: test data generation and application of a test data adequacy criterion. A test data generation technique is an algorithm that generates test cases, whereas an adequacy criterion is a predicate that determines whether the testing process is finished [4].

Web applications (WebApps) are being used extensively in business, social, organizational, and governmental functions. The continual availability of WebApps is one of the advantages to use them by many users without regard to their location or time limitations [5]. However, this demands high reliability of WebApps. Inadequate testing poses huge risks including downtime and loss of users' trust and convenience. The Web's ubiquity and users' reliance on it have made it crucially important to ensure the quality, correctness, and security of WebApps.

The process of constructing or choosing test data manually to cover a certain testing criterion requires experience and time. Automating the process of test data generation can solve these issues. One possible solution is generating test data at random, which can speed up the process significantly. However, there is no guarantee that the generated data will cover the testing criterion, or the data will be useful in error detection. Therefore, a more sophisticated and intelligent technique is required. Search-based software testing formulates testing as an optimization problem, which can be solved using computational search techniques from the field of Search Based Software Engineering.

This paper presents an automated test data generation approach based on genetic algorithm (GA) for data flow testing of ASP.NET webApps. The paper is organized as follows: Section 2 presents the related work. Section 3 briefly covers the concepts of GAs and some of the used utilities. Section 4 presents the proposed test data generation approach. Section 5 presents a case study. Section 6 presents the experimental results. Section 7 presents the conclusion of this work.

Manuscript received August 29, 2019; revised February 3, 2020

Islam T. Elgendy and Adel A. Sewisy are with the Department of Computer Science, Faculty of Computers and Information, Assiut University, Egypt. e-mail: islam.elgendy@aun.edu.eg

Moheb R. Girgis is with the Department of Computer Science, Faculty of Science, Minia University, Egypt.

II. Related Work

As this paper focuses on automatic test data generation for webApps using GA, this section reviews some of the research work in the field of search-based test data generation and webApp testing. Sharma et al. [6] presented a survey of GA approach for addressing the various issues encountered during software testing, and discussed the applications of GA in different types of software testing. They concluded that using GA, the results and the performance of testing can be improved. Li et al. [3] presented a broad survey of testing advances in webApps and discussed the techniques employed, targets, goals, inputs/outputs and stopping criteria. They stated that there are various testing goals, such as ensuring testing adequacy, finding faults, etc. and the choice of a testing technique depends on the testing goal. Lakshmi and Mallika [7] presented a comparative study of some of the techniques, prominent tools, and models for WebApp testing, and highlighted the research directions for some of the WebApp testing techniques.

Girgis [8] presented an automatic test data generation technique based on GA, which is guided by the data flow dependencies in the program, to search for test data to fulfil the all-uses criterion. The input to the GA is an instrumented version of the program under test, number of input variables, the domain and precision of each input variable, and list of definition-use (def-use) associations. The output of the GA is a set of test cases and the list of def-use associations covered by each test case, and a list of uncovered def-use associations, if any.

Alshahwan and Harman [5] adapted a set of related search based testing algorithms for webApp testing, and implemented an automated test data generation approach for PHP

webApps. Setiadi et al. [9] proposed a method to reduce the number of test cases for detecting errors in a concurrent program. They used data flows and branch structure to reduce redundant test cases, identifying only the interleavings that affect the branch outcomes rather than identifying all interleavings affecting shared variables. Setiadi et al. [10] further improved their work by analyzing data dependency to generate test cases affecting sequences of locks and shared variables, reducing the required memory space. Also, they generated test cases for detecting race conditions caused by accesses through reference variables. Furthermore, by exploiting previous test results, they reduced the effort for checking race conditions.

Takamatsu et al. [11] extended the Seeker tool developed by Thummalapenta et al. [12], which is based on branch coverage, focusing on multiple targets, identifying all the involved targets in uncovered branches and evaluating method sequences according to a fitness function, while applying a search strategy to suppress combinatorial explosion. Boussasa [13] introduced the use of Novelty Search (NS) algorithm to the test data generation problem based on statement-coverage criteria. The NS adaptation attempted to exploit the large search space of input values and catch relevant test cases that may cover as much as possible the executed statements.

Girgis et al. [14] have presented an approach to data flow testing of WebApps. They presented an approach that includes the construction of a WebApp data flow model to aid WebApps data flow analysis. In this approach, testing is conducted in four different levels, Function, Inter-procedural, Page, and Inter-Page levels. In each level, the def-use pairs of the variables are obtained. Then, selecting test data that cover these def-use pairs ensures the fulfilment of the all-uses criterion.

Akhter et al. [15] proposed a GA based-test data generation technique to achieve path coverage of the program under test. The proposed technique is a multiple population algorithm, in which small changes are performed and the populations are combined to find the fittest test data. The fittest solutions from all the populations form a new population. Crossover is performed to have a new population which is also combined with the fittest population and the process continues. After all iterations, the set of fittest test data is obtained for maximum path coverage. Azam et al. [16] presented a toolkit for automated test data generation, and test case prioritization. They used GA and fuzzy rule based system to generate automated test data with focus on boundary values testing.

Scalabrino et al. [17] presented a search-based tool, called Ocelot, for the automatic generation of test cases in C. Running Ocelot consists of two distinct macro-phases: build and run. In the build phase, the target program is instrumented, and then compiled. The output of such a phase is a static library, which is linked by the tool and used in the run phase, where a search-based algorithm is exploited to identify a set of inputs that maximize the code coverage. The output of this phase is a set of test data that can be used to test the target program.

As far as the authors are aware, none of these researches have used GAs to generate test data for data flow testing of webApps.

III. Genetic Algorithm Concepts and Utilities

This section presents the basic concepts of genetic algorithms (GAs), and some utilities for automated testing, namely Microsoft Coded User Interface Testing (CUIT) and Genetic Algorithm Framework (GAF).

A. Genetic algorithms

The basic concepts of GAs were developed by Holland [18]. GA is a search-based optimization technique inspired from the natural search and selection processes that leads to the survival of the fittest individuals. GAs generate a sequence of populations by using a selection mechanism, and use crossover and mutation as search mechanisms [19]. The idea behind GAs is that they construct a population of individuals represented by chromosomes, which is normally a character string similar to the DNA chromosomes. These chromosomes represent solutions to a problem. The chromosomes then evolve according to the selection, mutation and reproduction rules. The fitness of each individual (chromosome) in the environment is measured. Individuals with high fitness values in the population are selected by reproduction, and a new population is derived through crossover and mutation, in which individuals may have better fitness in their environment. In crossover two chromosomes swap genetic information as in the process of sexual reproduction. Mutation makes slight modifications in a small part of the population that represents an evolutionary step. The structure of a simple GA is given in algorithm 1.

Algorithm 1: Basic GA

```
1 begin
2     initialize population;
3     evaluate population;
4     while termination criterion not reached do
5         select solutions for next population;
6         perform crossover and mutation;
7         evaluate population;
8     end
9 end
```

The termination criterion of the algorithm can be either reaching a solution to the problem, or reaching the maximum number of iterations, meaning that a solution cannot be found given the available resources.

B. CUIT and GAF

In Visual Studio, automated tests that drive an application through its user interface (UI) are known as coded UI tests (CUITs) [20]. They are frequently used to verify that the application is working correctly including its user interface, and to automate an existing manual test. CUIT is used to run the webApp with certain data generated from the proposed tool to test the viability of the test data and the achieved coverage.

The Genetic Algorithm Framework (GAF) is a .Net/Mono assembly, freely available via NuGet, that allows a GA based solution to be implemented in C# [21]. The population, which is the collection of possible solutions, can be initialized randomly, or a set of starting possible solutions can be

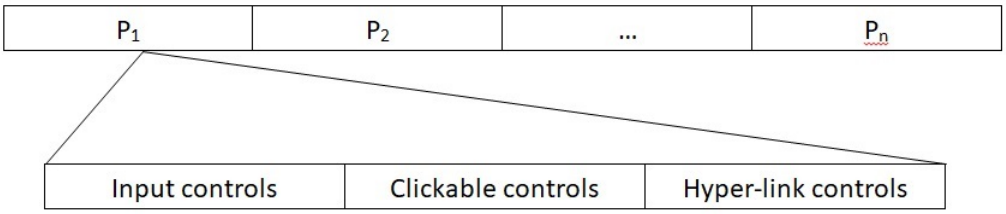
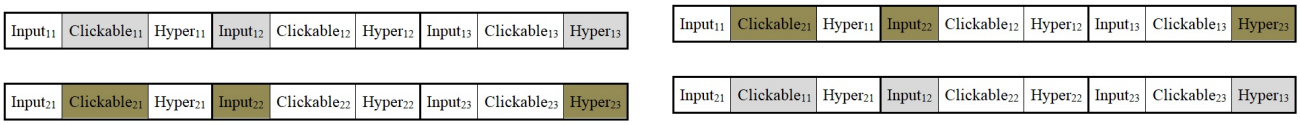


Fig. 1: The constructed chromosome and the three groups within each page.



(a) Before Block-crossover.

(b) After Block-crossover.

Fig. 2: Two possible chromosomes before and after the Block-crossover.

added to the population. The GAF supports binary, integer, real and object-based genes (i.e. a chromosome with objects as genes). The GAF allows the definition of any fitness function as well as a termination function, which causes the run to stop. It also provides many genetic operators, such as: crossover and mutation, and supports adding new custom genetic operators.

IV. Proposed Approach

This section describes a proposed approach for data-flow test data generation for ASP.NET webApps. In an early work, the authors [22] presented a tool for automated data flow testing of ASP.NET webApps, where the webApp under test is instrumented and analyzed to compute the def-use pairs. This paper extends this work by using a GA for generating test cases automatically to cover the computed def-use pairs.

A. Representation

The chromosome is constructed from the generated report of the static analysis [22]. Every user interface (UI) control is considered as a gene in the chromosome. Figure 1 shows the constructed chromosome, where P_1, P_2, ..., P_n represent the pages of the webApp under test. The web pages in the webApp under test are added to the chromosome one by one. Within each page, the UI controls are added in a certain order. Three groups of controls may exist within each page: Input controls (such as *textboxes*, or *comboBoxes*), clickable controls (such as *radioButtons*, *checkboxes*, or *buttons*), and hyper link controls (including *linkButtons*). Within the clickable controls block, the buttons are grouped at the end of the block. The three groups within each web page are shown in figure 1.

The reason behind this chromosome structure is that we want to generate first data for input controls that might be used in the code-behind file, then click on controls that won't cause page transfer, and finally choose one of the hyper-links to transfer to another page. This procedure is repeated until the chromosome is completed. The GAF framework [21] allows us to construct the chromosome genes as objects rather than binary strings. This means that every gene in the chromosome is a UI control object. Thus, novel crossover and mutation operators are developed to handle this chromosome representation. The initial population is constructed with random values for the input controls, random clicking order for the clickable controls, and one hyperlink is chosen at random. The tester is required to specify the starting page of the webApp from which the test starts, and the type of each textbox value (text or numeric) and the range of possible values for numeric data, or the string length for text fields.

B. Genetic operators

As explained before, the chromosome is not a binary string but rather a collection of UI control objects, with each control is considered to be a gene. Therefore, novel crossover and mutation operators are developed to manipulate the chromosome. These novel operators are called ***block crossover*** and ***control-based mutation***.

1) Block crossover: The chromosome is partitioned as web pages. The crossover is performed between two chromosomes page-wise. One of the three blocks within the web page is chosen randomly and swapped between the two chromosomes. For example, let's assume that we have three web pages in the application. Two possible chromosomes are shown in figure 2a. Assume that for the first page block 2 is chosen at random, for the second page block 1 is chosen, and for the third page block 3 is chosen. The generated chromosomes are shown in figure 2b.

2) Control-based mutation: The traditional mutation is not suitable because the chromosome is not a binary string but consists of UI controls as genes. Thus, for each control type, there is a different mutation operation. Input controls mutation involves generating a new value randomly within the range specified by the tester. Clickable controls mutation involves flipping the value (clicked to unclicked, or unclicked to clicked). Button controls mutation involves changing the order of the button clicks. Hyperlink controls mutation involves choosing another hyperlink randomly.

C. Fitness function

The fitness function is performed based on the coverage percentage for the solution. Each chromosome ch is decoded into a set of test data, which represents the solution, and the

instrumented webApp is run with this set of data. Running the instrumented webApp produces a file containing the traversed path. This path is checked with the list of def-use pairs, and its fitness is evaluated as follows:

$$f(ch) = \frac{number\ of\ covered\ def-use\ pairs}{total\ number\ of\ def-use\ pairs} \quad (1)$$

$f(ch)$ is the fitness of the chromosome ch. A test case which is represented by the chromosome ch is considered effective if its fitness value $f(ch) > 0$. In each generation, every solution in the current population is run and the fitness is calculated. The best solution, which achieves the highest coverage percentage, is selected as a test case. The population goes through block crossover and control-based mutation to produce a new population. The new population is evaluated using the same procedure while keeping the last coverage percentage to achieve a better accumulated coverage percentage. To illustrate this, let's assume that we have a total of 50 def-use pairs in a webApp. The first generation was evaluated and the best solution covered 20 def-use pairs. Hence, the coverage is 40%. Now, the new generation will keep that 40% coverage in the evaluation. Assume that the next solution covered 5 more def-use pairs, which means that with this solution the total number of covered def-use pairs became 25, so the accumulated coverage is now 50%. This procedure is repeated until a full coverage is achieved or the maximum number of generations is reached.

Algorithm 2: A GA algorithm to automatically generate test cases for a given webApp.

input : Instrumented version P' of the webApp to be tested P;
List of def-use pairs to be covered;
Input controls related information;
Population size (Pop_size);
Maximum no. of generations (Max_Gen);
Empty input probability;
Starting url address;
output: Set of test cases for P, and the set of def-use pairs covered by each test case;
List of uncovered def-use pairs, if any;

begin
  Step 1: Initialization
    Initialize the def-use coverage table to uncovered;
    Create $Initial\_Population$;
    $Cur\_population = Initial\_Population$;
    Set of test cases for P, $S = \phi$;
    $Acc\_Coverage\_Percent = 0$;
    $No\_Of\_Generations = 0$;
  Step 2: Generate test cases
    while $Acc\_Coverage\_Percent \neq 100$ *and* $No\_Of\_Generations \leq Max\_Gen$ **do**
      $best = Evaluate\_Population(Cur\_population)$;
      $Cur\_Coverage = checkCoverage(best)$;
      if $Cur\_Coverage > Acc\_Coverage\_Percent$ **then**
        $Acc\_Coverage\_Percent = Cur\_Coverage$;
        $S = S \cup best$;
        Update the def-use coverage table;
      end
      Create $New\_Population$ using block-crossover and control-based mutation operators;
      $Cur\_Population = New\_Population$;
      Increment $No\_Of\_Generations$;
    end
  Step 3: Produce output
    Return S, and set of def-use pairs covered by each test case;
    Report on uncovered def-use pairs, if any;
end

Algorithm 3: Population evaluation.

input : The population Pop;
The $pageList$ data structure;
output: Best solution of the population;

begin
  $pageIndex = FindStart(Pop, url)$;
  for *every chromosome ch in Pop* **do**
    Decode every gene into a UI control object;
    Initialize values of the array $visited$ to false;
    Create an empty coded UI file ($CUIF$);
    $startIndex = pageList[pageIndex].start$;
    $endIndex = pageList[pageIndex].end$;
    $visited[pageIndex] = true$;
    for *every UI control uic in ch from the* $startIndex$ **to** $endIndex$ **do**
      Write coded UI command into $CUIF$ based on the type of the uic object to perform the corresponding action on this UI control;
      if $uic.Type = hyperlink$ **then**
        $targetPage = uic.Target$;
        $tInd = search(targetPage, pageList)$;
        if $tInd \neq -1$ *and* $!visited[tInd]$ **then**
          $startIndex = pageList[tInd].start$;
          $endIndex = pageList[tInd].end$;
        else
          close $CUIF$;
          $break$;
        end
      end
    end
    Compile and run $CUIF$ to automatically run the webApp with the generated test data;
    Calculate $f(ch)$ based on the percentage of the covered def-use pairs by the traversed path;
  end
  Return the best chromosome that has the highest coverage.
end

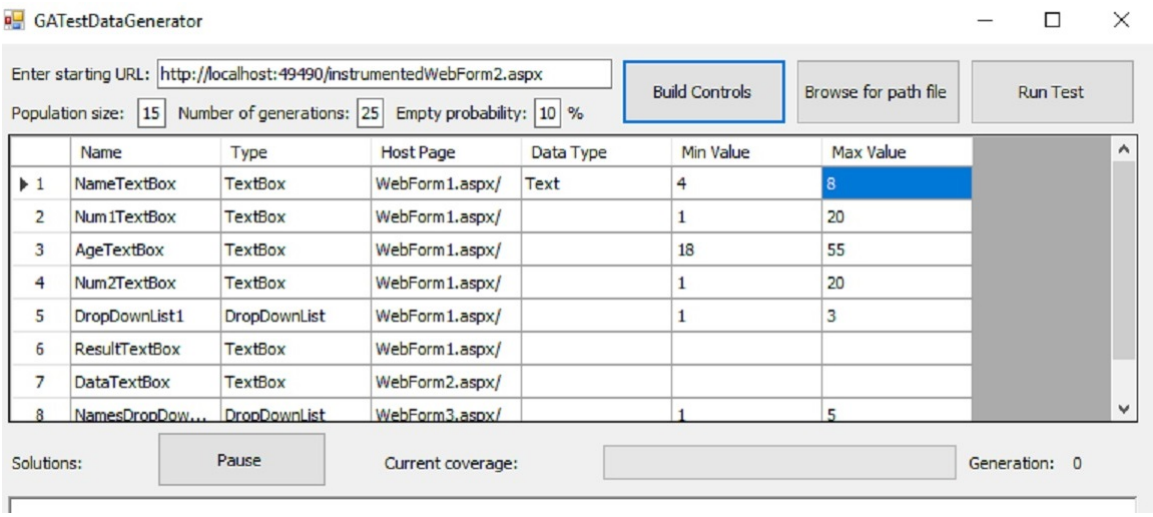


Fig. 3: The GA test data generator form.

D. The Overall GA Algorithm

Algorithm 2 shows the steps of the proposed GA algorithm ($GATestDataGenerator$). The algorithm accepts as input an instrumented version of the webApp to be tested, the list of def-use pairs to be covered, the input controls related information (type and range for each input control). Also, it accepts the starting URL of the webApp, population size, maximum number of generations, and the empty probability (the probability that no input controls are used and buttons are clicked right away) . The algorithm produces a set of test cases, the set of def-use pairs covered by each test case, and the list of uncovered def-use pairs, if any.

The algorithm uses a table, called the def-use coverage table, to record the traversed def-use pairs. In this table, each element corresponds to a def-use pair. Initially, all pairs are marked as uncovered. A set of final test cases S is initially empty, and the GA population is created initially at random. Line 11 calls Algorithm 3 (which is explained in the next subsection) to evaluate the population and returns the resultant best solution. Then this best solution is checked for coverage in line 12. If the coverage of the best solution increases the accumulated coverage, that solution is added to the set S as one of the final test cases, the accumulated coverage is set to the best coverage, and the def-use coverage table is updated. The number of generations is incremented, and the new population is created using block-crossover and control-based mutation operators. The process is repeated until a full coverage is achieved or the max number of generations is reached. The set S is returned, and a report that includes the def-use pairs covered by each test case and the list of uncovered def-use pairs, if any, is displayed.

E. Running tests

In order to evaluate the fitness of the solution, we use the algorithm $Evaluate\_Population$ (Algorithm 3). In this algorithm, the chromosome is decoded into a set of test data and run automatically on the webApp using Microsoft coded UI test. This requires generating a coded UI file based on the current chromosome, and the file is compiled and run automatically to perform the test on the webApp. Running the instrumented webApp produces a text file that contains the line numbers of the traversed path, which is used to check the fitness of the chromosome as explained before. We use a data structure named *pageList* to track the start and end indices of every web page and its internal blocks within the chromosome. The *pageList* is used to retrieve information about the positions of the UI controls of the web pages. Also, we use an array of Boolean values named *"visited"* to keep track of the status of the web pages. The value true means a web page is visited, false otherwise.

V. Case Study

This section presents a case study to show how the tool generates automatically test data to cover the def-use pairs of a webApp. The webApp used in the case study consists of three web pages named "webForm1.aspx", "webForm2.aspx", and "webForm3.aspx". The code and description of the webApp is provided on GitHub [1].

The initial static analysis showed that the webApp has a total of 230 statements, 82 variables, 36 anomalies, and 91 def-use pairs. Figure 3 shows the GA test data generator form, where the tester enters the starting URL page from which the test will run, the population size, the number of generations, and the empty probability. This is the probability of creating a solution without any data for the input controls and directly clicking on the buttons and hyper-link. The form also shows the input controls from the analysis phase, and for each input control the tester determine the type of the data in the text box whether it is text or by default numeric and the range of the data in the input control. In case of the text type, the range indicates the possible length of the input string. The characters themselves are generated at random. For list controls, the range determines the selected index that might be chosen for the control. If the control is left without any data, this means that the tool will not generate any data for this control.

Once the tester specifics all the required data, he/she clicks on the *"Build Controls"* button, and then browses for the text path file that will be used to check the coverage of the current test run. Finally, the tester clicks on the *"Run Test"* button that will automatically start generating data for the webApp

[1] https://github.com/islamelgendy/webApplication/tree/master

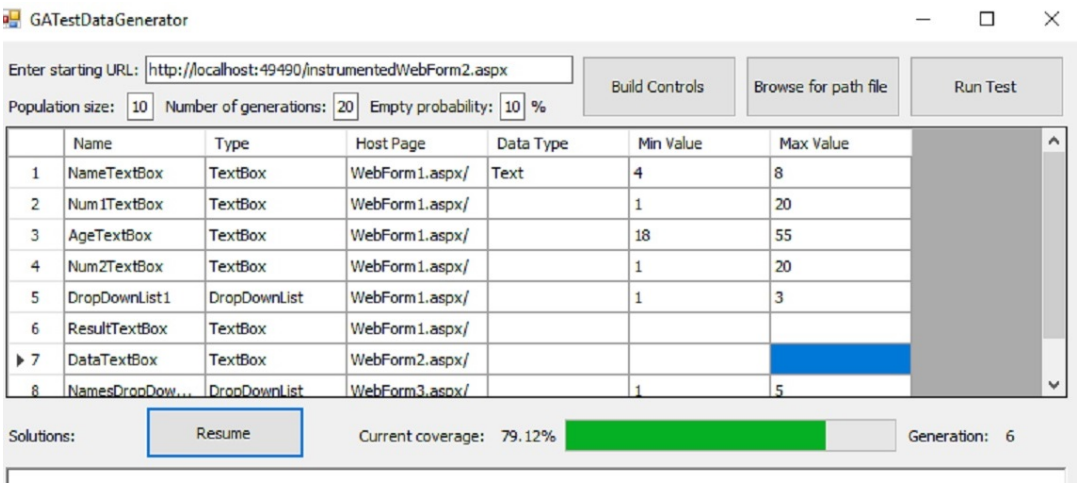


Fig. 4: During the run.

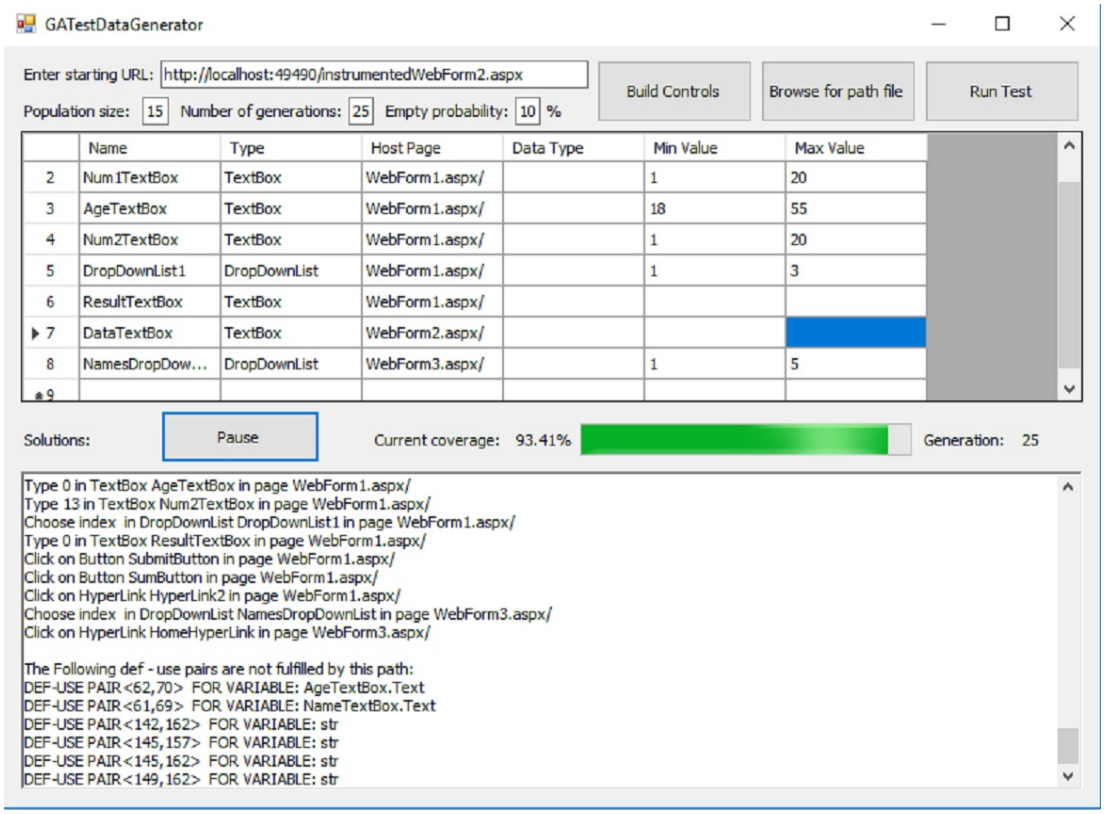


Fig. 5: After the finish of the run.

and evaluates each run in terms of coverage. In the duration of generating test data, the tester will not have control over the computer, but he/she can only pause the process at any time and then resume it again. After each generation, the best solution is selected and the tester will be notified with the current coverage in the progress bar as shown in figure 4. The GA algorithm will finish once it satisfies the stopping criteria. The final solutions will be displayed in the bottom text area showing the coverage percentage reached with every test case and in which generation. Also the unfulfilled def-use pairs left will be shown. In this example, the tool managed to find an accumulated coverage of 93.41% with only 6 def-use pairs left uncovered, 2 of which are infeasible. The end results and parameters used in this run are shown in figure 5.

VI. Experimental Evaluation

Two types of experiments have been conducted. The purpose of the first one is to evaluate the ability of the test data generated by the proposed GA to cover the def-use pairs of the webApp being tested (i.e. fulfils the all-uses criterion). The purpose of the second one is to evaluate the effectiveness of the generated test data in exposing webApp errors. The materials for these experiments were five different webApps [2] [3] [4].

[2]https://github.com/islamelgendy/CS651_TEST , https://github.com/islamelgendy/MySessions

[3]https://code.msdn.microsoft.com/Getting-Started-with-221c01f5

[4]https://www.codeproject.com/Articles/1575/ASP-NET-To-Do-List-Application

TABLE I: Empirical evaluation of the def-use coverage of the test data generated by proposed GA approach.

Web Application	# web pages	# def-use pairs	Method	Coverage %	% infeasible def-use pairs	# of test cases	generation # for the last test case
UIControls_Test	3	91	GA	93.40%	2.20%	7	23
			RT	86.81%		12	21
ToDo list	2	118	GA	71.19%	8.47%	5	14
			RT	65.25%		6	13
Wingtip toys	6	67	GA	56.72%	4.48%	4	4
			RT	47.76%		3	18
CS651_TEST	1	45	GA	84.44%	15.56%	3	3
			RT	84.44%		3	5
Sessions	2	28	GA	85.71%	7.14%	2	2
			RT	76.92%		2	2

TABLE II: The number and percentage of discovered errors of each type. The error code 'C' stands for "Computation errors", the error code 'D' stands for "Domain errors", the error code 'E' stands for "Event errors", and the error code 'P' stands for "Presentation errors"

Error Code	Error Freq.	# discovered errors	% of discovered errors
C1	5	4	80%
C2	1	1	100%
C3	18	10	55.56%
C4	21	19	90.48%
C5	4	3	75%
C6	1	1	100%
Subtotal	**50**	**38**	**76%**
D1	16	16	100%
D2	2	2	100%
D3	4	3	75%
D4	3	3	100%
D5	8	5	62.5%
D6	1	1	100%
Subtotal	**34**	**30**	**88.24%**
O2	1	1	100%
O3	12	9	75%
O4	6	6	100%
O6	8	6	75%
O7	4	0	0%
Subtotal	**31**	**22**	**70.97%**
E1	3	3	100%
E2	4	4	100%
Subtotal	**7**	**7**	**100%**
P1	20	20	100%
P2	3	3	100%
P3	4	3	75%
P4	2	2	100%
P5	9	7	77.78%
P6	2	2	100%
Subtotal	**40**	**37**	**92.5%**
Total	**162**	**134**	**82.72%**

TABLE III: The number and percentage of discovered errors in webApps.

Web Application	# of seeded errors	# of detected errors	% of detected errors
UIControls_Test	40	38	95.00%
ToDo list	54	44	81.48%
Wingtip toys	23	10	43.48%
CS651_TEST	24	22	91.67%
Sessions	21	20	95.24%
Total	**162**	**134**	**82.72%**

In the first experiment, the proposed GA technique was compared with random testing (RT) technique. In order to have fair comparison, the random test data generator was designed to randomly generate a number of test cases equals to the maximum number of generations of the proposed GA. The population size was set to 15, and the number of generations was set to 25, for all applications. In addition, the block-crossover probability was set to 0.8, the mutation probability was set to 0.08 for the input and clickable controls, and the mutation probability was set to 0.64 for the hyperlink controls. All these parameters were experimentally determined. For each webApp, we applied both techniques ten times, and reported the average results. Table I shows the results of the first experiment. In this table, column 1 shows the name of the webApp under test, column 2 shows the number of web pages in the webApp, column 3 shows the number of def-use pairs in the webApp, column 4 shows the used method, column 5 shows the coverage percentage of the generated data using the proposed GA and RT methods, column 6 shows the percentage of the infeasible def-use pairs, column 7 shows the number of test cases required to achieve the reported coverage percentage, and the last column shows the generation/iteration number where the last test case occurred. For webApp $UIControls\_Test$, the proposed GA technique uncovered only 6.6% (6 pairs) out of them there were two infeasible pairs, and the last test case occurred at generation number 23. While the RT technique uncovered only 13.19% (12 pairs), and the last test case occurred at iteration number 21. All of the def-use pairs of $CS651\_Test$ webApp were covered, by both techniques, apart from the 7 infeasible def-use pairs in both techniques.

It is worth noting that, in small webApps the performance of both techniques are close, like in $CS651\_Test$ webApp and $Sessions$ webApp. However, in more complex and large applications, the performance of the proposed GA is better than RT in terms of both coverage percentage and the number of used test cases to cover the reported coverage percentage.

In the second experiment, a number of errors were seeded manually in the five webApps, and the tool was applied to the erroneous versions of the applications. The errors, which may occur in webApps, fall into five categories: computation errors, domain errors, object-oriented errors, event errors, and presentation errors. The first four groups, [23] are related to the code-behind file, while the last group, [24]. is related to the ASPX file. The output of the executions (actual output) were compared with the correct output (expected output), and the static and dynamic reports produced by the tool were studied. If the error showed up either by some deviation in the output, or by one of the messages of the report, then this meant that the generated test data which fulfilled the all-uses criterion have enabled the discovery of the error. Table II shows the number and percentage of discovered errors of each type. From this table, it can be observed that, using the generated test data, 76% of the computation errors were discovered, 88.24% of the domain errors were discovered, 70.97% of the object-oriented errors were discovered, 100% of the event errors were discovered, and 92.5% of the presentations errors were discovered. A total of 82.72% of the errors were discovered using the generated test data. Table III shows the number and percentage of discovered errors in the five webApps. In most cases, it is observed that the higher the def-use coverage percent achieved for the webApp, the higher the errors were detected.

VII. Conclusion

This paper presented a GA-based approach and a supporting tool for data-flow test data generation for ASP.NET webApps. The proposed tool accepts as input the webApp under test, instruments it, and statically analyses it to compute the def-use pairs. In the proposed GA, each gene in the chromosome is a UI control object. Two novel GA operators, block crossover and control-based mutation operators, were developed to manipulate such chromosome.

In each generation, every solution in the population is evaluated, and the best solution, which achieves the highest coverage percentage, is selected as a test case. This process is repeated until the best accumulated coverage percentage is achieved. The tool produces the combined set of test cases, the set of def-use pairs covered by each test case, and a list of uncovered def-use pairs, if any. A case study was presented to illustrate how the tool works. In order to evaluate the effectiveness of the generated test data in fulfilling the all-uses criterion and exposing webApp errors, an empirical evaluation was performed. From the experiments it can be concluded that: the higher the def-use coverage percent achieved for the webApp, the higher the errors were detected.

References

[1] B. Korel, *"Automated software test data generation,"* IEEE Transactions on Software Engineering, vol. 16, pp 870-879, 1990.

[2] J. Edvardsson, *"A Survey on Automatic Test Data Generation,"* In Proceedings of the 2nd Conference on Computer Science and Engineering, pp 21-28, 1999.

[3] Y. F. Li, P. K. Das, and D. L. Dowe, *"Two Decades of Web Application Testing - A Survey of Recent Advances,"* Information Systems, vol. 43, pp 20-54, 2014.

[4] P. G. Frankl, S. N. Weiss, *"An Experimental Comparison of The Effectiveness of Branch Testing and Data Flow Testing,"* IEEE Transactions on Software Engineering, vol. 19, no. 8, pp 774-787, 1993.

[5] N. Alshahwan, and M. Harman, *"Automated Web Application Testing using Search Based Software Engineering,"* In 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 3-12, 2011.

[6] C. Sharma, S. Sabharwal, R. Sibal. *"A Survey on Software Testing Techniques using Genetic Algorithm,"* International Journal of Computer Science Issues (IJCSI), vol. 10, no. 1, pp 381, 2013.

[7] D. R. Lakshmi, and S. S. Mallika. *"A Review on Web Application Testing and its Current Research Directions,"* International Journal of Electrical and Computer Engineering (IJECE) Vol. 7, No. 4, pp. 2132-2141, August 2017.

[8] M. R. Girgis, *"Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm,"* Journal of Universal Computer Science, vol. 11, no. 6, pp 898-915, 2005.

[9] T. E. Setiadi, A. Ohsuga, and M. Maekawa. *"Efficient Execution Path Exploration for Detecting Races in Concurrent Programs,"* IAENG International Journal of Computer Science, vol. 40, no. 3, pp 143–163, 2013.

[10] T. E. Setiadi, A. Ohsuga, and M. Maekawa. *"Efficient Test Case Generation for Detecting Race Conditions,"* IAENG International Journal of Computer Science, vol. 41, no. 2, pp 112–130, 2014.

[11] H. Takamatsu, H. Sato, S. Oyama, and M. Kurihara. *"Automated Test Generation for Object-Oriented Programs with Multiple Targets,"* IAENG International Journal of Computer Science, vol. 41, no. 3, pp. 198–203, 2014.

[12] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. *"Synthesizing Method Sequences for High-Coverage Testing,"* ACM SIGPLAN Notices, vol. 46, no. 10, pp 189–206, 2011.

[13] M. Boussaa, O. Barais, G. Sunyé, B. Baudry. *"A Novelty Search Approach for Automatic Test Data Generation,"* In 2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing, pp 40-43, 2015.

[14] M. R. Girgis, A. I. El-Nashar, T. A. Abd El-Rahman, and M. A. Mohammed. *"An ASP.NET Web Applications Data Flow Testing Approach,"* International Journal of Computer Applications, vol. 975, pp 8887, 2016.

[15] N. Akhter, A. Singh, G. Singh, *"Automatic Test Case Generation by using Parallel 3 Parent Genetic Algorithm,"* International Journal for Research in Applied Science and Engineering Technology, vol. 6, pp 114-121, 2018.

[16] M. Azam, K. Sultan, S. Dash, S. Naqeeb, M. Alam, *"Automated Test-case Generation and Prioritization Using GA and FRBS,"* International Conference on Advanced Informatics for Computing Research, pp 571-584, 2018.

[17] S. Scalabrino, G. Grano, D. Di Nucci, M. Guerra, A. De Lucia, H. Gall, R. Oliveto, *"OCELOT: A Search-Based Test-Data Generation Tool for C"*, In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp 868-871, 2018.

[18] J. Holland, *"Adaptation in Natural and Artificial Systems: An Introductory Analysis with Application to Biology,"* Control and artificial intelligence, 1975.

[19] M. Srinivas, L. M. Patnaik, *"Genetic Algorithms: A Survey,"* IEEE Computer, vol. 27, no. 6, pp 17-26, 1994.

[20] https://docs.microsoft.com/en-us/visualstudio/test/use-ui-automation-to-test-your-code?view=vs-2017, April 2019.

[21] https://bitbucket.org/johnnewcombe/gaf/wiki/Home, April 2019.

[22] I. T. Elgendy, M. R. Girgis, A. Seiwsy, *"An Automated Tool for Data Flow Testing of ASP.NET Web Applications,"* Applied Mathematics & Information Sciences, vol. 14, no. 4, pp 679-691, 2020.

[23] L. J. White and E. I. Cohen. *"A Domain Strategy for Computer Program Testing,"* IEEE Transactions on Software Engineering, vol. 6, no. 3, pp 247-257, 1980.

[24] N. Mansour and M. Houri, *"Testing Web Applications,"* Information and Software Technology, vol. 48, no. 1, pp 31–42, 2006.