A Pseudo-Genetic Algorithm for Optimising Test Cases

Suhaila M. Yasin, *Member, IAENG*, Paul A. Strooper, and Jim R. H. Steel

***Abstract*— Genetic Algorithms have been widely used to generate test cases by automatically searching a space for suitable solutions to a search problem. In this paper, we intend to produce a set of test cases that effectively find faults in a web application from an initial set of test cases. We introduce a variant of genetic algorithms, a pseudo-genetic algorithm that generates model-based test cases and transforms them into executable test cases. We incorporated the pseudo-genetic algorithm in a search-based testing tool called *MutateIFML*. We included the web application's model during the optimisation process to ensure that the offspring test cases are syntactically sound. Then, we evaluate the effectiveness of the test cases using the mutation score of the test cases. We found that *MutateIFML* performs better on a collection of mutant systems under test that were created by hand, based on bug reports of a web application, compared to mutant systems under tests created by a mutation testing tool. Finally, we discuss further improvements for *MutateIFML*.**

***Index Terms*— Search-based testing, model-based testing, functional testing, genetic algorithms**

I. INTRODUCTION

GENERATING test cases is challenging, particularly when it requires automating the process to produce executable test cases. According to Utting [1], the process usually involves initialising the system under test (SUT), putting the SUT in the required context, creating the test input values, passing those inputs to the SUT, recording the SUT response, comparing that response with the expected outputs, and assigning a pass/fail verdict to each test. Researchers have implemented various approaches to address these challenges, depending on the type of problem that they want to solve, and the type of SUT involved. The application of Genetic Algorithms (GAs) is quite prevalent in test case generation that involves test case optimisation. GAs are applied as part of the search-based testing (SBT) approach, which uses metaheuristic search techniques [2] to search for the best solution in a search space or population.

Sometimes, GAs are modified to solve specific search problems, such as the Memetic Algorithms that incorporates local search operators into GAs [3]. In this paper, we use the term 'fitness' to refer to the performance of both the parent and offspring test cases. We also use the term 'population' to refer to the set of test cases. We modify GAs with several intentions. Firstly, the initial selection phase of GAs is used to evaluate the fitness of the initial set of test cases. The fitness of these test cases later provides valuable insight regarding the limitations of the initial population, specifically concerning the nature of faults in the SUT that were not discovered by the test cases. Secondly, we use the Interaction Flow Modelling Language (IFML) to represent candidates, as opposed to traditional binary string representation. IFML has the capability to model the content, user interaction and control behaviour of the front ends of WAs [4]. We want to produce multiple test cases with varied interaction flows, and the IFML elements provide a suitable representation for the model-based test cases that we are using. Thirdly, the termination of GAs can be configured quite easily to satisfy specific search problems. In our case, we want to increase the effectiveness of the test cases in revealing faults in the SUT.

Testing web applications (WAs) manually is difficult. Today's fast-paced WA development practices often cause testing to be neglected [5]. Automating the test case generation and optimisation increases the efficiency of testing. We have developed a prototype SBT support tool called *MutateIFML*. *MutateIFML* uses the modified GA to generate new test cases from the existing pool of test cases. *MutateIFML* also facilitates test execution by controlling Humbug [6] a mutation-testing tool that executes test cases by using PHPUnit and Selenium WebDriver[1]. It evaluates the fitness of the test cases by calculating the mutation score of these test cases. We used two groups of mutant SUTs: (1) mutant SUTs that are automatically generated by Humbug and (2) mutant SUTs produced by hand based on a fault taxonomy for WA's faults, and a bug report of the SUT. The former is produced at code level, whereas the latter is performed at a higher level to achieve more diversity in terms of the mutant classification.

The paper is set up as follows. In Section II, we review the related literature on SBT, including the use of GAs in SBT. In Section III, we present our modified GA. In Section IV, we describe the SBT support tool, *MutateIFML* that we developed. In Section V, we explain the validation of *MutateIFML*'s prototype that was performed with *OpenBiblio*, a WA library management system. In Section VI, we discuss the results from the validation of *MutateIFML*. In Section VII we elaborate the plan for improving *MutateIFML*.

Manuscript received December 22, 2016; revised January 16, 2017.

Suhaila M. Yasin is with the School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane 4072, Queensland Australia, on leave from Universiti Tun Hussein Onn Malaysia, 86400 Parit Raja, Batu Pahat, Johor, Malaysia (e-mail: s.mohdyasin@uq.edu.au).

Paul A. Strooper is with the School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane 4072, Queensland Australia. (e-mail: pstroop@itee.uq.edu.au).

Jim R. H. Steel is with the Australian e-Health Research Centre, CSIRO, Boulder, Herston 4029, Queensland Australia. (e-mail: jim.steel@csiro.au).

[1] Selenium WebDriver - http://www.seleniumhq.org/projects/webdriver/

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

II. Related work

This research project builds on earlier work from three areas: search-based testing, model-based testing (MBT) and fault-seeding. This section discusses the findings from earlier work within the scope of WA testing.

A. Search-based Testing

The term "search-based testing" is used interchangeably with "search-based software testing". In this paper, the following definition of SBT is used: "Applying a metaheuristic algorithm to a software testing approach in order to optimise a certain testing task that was in focus" [7]. The term "metaheuristic" here refers to "all stochastic algorithms with randomisation and local search" [8]. The randomisation causes the local search to be able to be moved to search on the global scale [8]. In software testing, SBT is applied with the intention of supporting automation as opposed to manual testing [7], and reducing testing effort, while at the same time being adaptable to changes [2].

Consider Evolutionary Algorithms (EAs) as an example of a search algorithm. The search algorithm begins with a group of individuals, which are known as the initial population. These individuals may represent various objects in testing (source code, test data, test cases, or classes), depending on the problem that a tester wishes to solve [9]. The search process is guided by the fitness function, a component that is capable of guiding the search towards revealing the potential solution within a practical time limit [7]. The fitness function helps the EAs in finding the best individual that could solve the problem, by measuring the individual's fitness value. If the best individual is not found, the search process will go through a repetitive cycle of selecting several individuals in the population, recombining their attributes (through operations such as crossover that switches attributes between a pair of individuals and/or the mutation of some attributes of selected individuals), and will then introduce the modified individuals as new members of the population (offspring). To achieve successful implementation of SBT, two elements of the SBT algorithms need to be accurately defined: representation of the search problem, and the fitness function that captures the objectives of the search problem and guides the search process [9].

SBT was introduced by Miller and Spooner with a simple technique for generating test data for floating-point inputs [10]. However, tool support for SBT remained elusive until several years later. The introduction of the GA and EA toolbox which helps automate the search process [11] led to further developments in SBT. This is evident in several later works that introduce the SBT approach with this toolbox to solve several software testing problems [12-14].

Most SBT contributions focus on test data generation [9]. Instead of optimising test data, we focus on optimising test cases to improve their effectiveness in testing. Control-flow coverage is the most commonly used criterion in evaluating the effectiveness of SBT algorithms [15]. Instead of control-flow coverage, we use mutation score as a criterion, since we are focusing our effort on searching solutions capable of discovering faults.

In the area of fault detection, academic contributions are limited [15]. Watkins et al. introduce a failure-pursuit strategy, based on a GA for system-level testing [16]. Tracey et al. propose the use of a Simulated Annealing (SA) algorithm to search for test data that reveal specification failures [17], demonstrated on three Ada programs. Recently, SA was compared with GAs and greedy algorithms in terms of test efficiency, *t*-way coverage and fault detection rate through combinatorial interaction testing performed on a collection of C and Java programs [18]. Learning from the flaws of GAs in previous implementations, Baudry et al. introduce Bacteriologic Algorithms to automate the test optimisation to increase the quality of the test suites [19]. Another method called the U-method is implemented by Guo et al. to trace faulty transitions in Finite-State Machines (FSM) [20]. Chan et al. demonstrate the use of EAs to discover unwanted behavior in a computer game's scenario [21]. The application of SBT is also reported in discovering faulty scenarios when simulating an automobile's parking, adaptive cruise control and anti-lock braking system [12, 14, 22]. In this paper, we apply GA, one of the most popular type of EA to automate the test optimisation of a WA. We aim to evaluate the effectiveness of the test cases in revealing faults related to the WA's interactions with its users.

B. Application of SBT in Web Application Testing

Several works extended SBT into WA testing. It has been suggested that SBT has a prominent role in future Internet applications, since SBT algorithms are adaptive to emerging challenges [23]. Extending previous research of SBT into WA testing presents challenges in terms of defining the representation of the candidates and the fitness function. The dynamic structure of web interfaces [24] also presents a difficulty. Since WAs are heavily event-driven [25], their response is influenced by the user's choice of actions. This causes the WA's interface to change with every interaction. WAs allow semantically long user interactions to accomplish complicated tasks, as long as the user session remains active. Marchetto et al. propose HILL, based on hill-climbing algorithms, to resolve the issue of maintaining the fault-detection effectiveness of long semantically interacting state-based sequences [26]. HILL is applied on the FSM of a WA called FSMWeb, and uses three fitness functions to find the most diverse test suite (i.e. the test suite that covers the most events in a WA) as the optimal solution. HILL compromises the length of the test case to find the most diverse test suite, by favouring a test suite containing limited test cases of different lengths over a huge suite containing test cases of smaller length. Manipulating longer test cases requires care as they involve higher computational costs, are difficult to interpret manually [27] and may lead to test case bloat [28]. Since WAs also allow shorter sequences of interactions, it is feasible to design a search algorithm that could address both kinds of interactions. We consider both kinds of interactions.

Alshahwan et al. [29] extend SBT algorithms [30, 31] and fitness functions from previous works [17, 32, 33] to develop a partially automated framework called the Search-Based Web Application Tester (SWAT) for the testing of WAs. Even though fault-detection capability is discussed, the approach leans towards achieving significant branch coverage and optimisation is done at the code-level. Our approach measures the effectiveness of the test cases in revealing seeded faults, and we perform optimisation at the

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

model level of the WA.

Another SBT tool called the Web Application Evolutionary Testing Tool (WETT) is proposed by Bolis et al. [34], in which GAs are used to optimise a collection of test cases expressed as Sahi [35] statements. The fitness function used in WETT values individuals with higher statement coverage and numbers of visited web pages. Although the Sahi statements provide a representation from the view of end-user interactions, the results showed low statement coverage and considerable time overhead. Our approach introduces a SBT tool that supports a high-level test case representation of a WA's behaviour, as opposed to the low-level representations commonly used in existing works. We use mutation score as our fitness function to value individuals that kill WA mutants.

A more recent work implements a GA to select potential locations that are most likely to contain failure events [36]. Instead of optimising test cases, Andrews et al. optimise test paths and types of actual failure events (defects). Further, the multi-objective fitness function used in their work emphasises these objectives: rewarding novelty of an offspring that is more distant from the existing population (exploring), and measuring the proximity between the offspring and the defects that have been discovered in the previous generations (prospecting or mining).

C. Application of MBT in Web Application Testing

For the purposes of MBT, a model is a representation of the SUT's behaviour [37]. As defined by Utting et al., there are four possible approaches for MBT [1]. Our approach uses the following interpretation of MBT: "the generation of executable test cases that include oracle information, such as the expected output values of the SUT, or some automated check on the actual output values to see if they are correct" [1].

To date, MBT in WA testing has been performed using C# [38], FSMs [39, 40], statecharts [25], Page Flow Diagrams [41], ATS [42], and ReWeb [43]. However, these modeling languages use a low-level representation of the SUT. Ernits et al. [38] focus on discovering errors in the actions of a WA. Using the NModel toolkit, smaller model programs are constructed in the C# programming language using an on-the-fly testing strategy. The drawback of this approach is the large overhead and difficulties experienced during the construction of the test harness. Aside from the NModel approach, other approaches [25, 39-43] are aimed at discovering inconsistencies in transitional paths. We use the model to develop a high-level representation of a WA. Since our work aims at revealing faults in the WA's behaviour, we focus on diversifying the user interactions of a WA by optimising its model-based test cases.

Recent modeling languages (IFML [44], IAML [45], and UWE [46]) have emerged to address the challenges of providing a high-level representation of a WA's interactions with the end-user. The standard UML diagrams lack precise modeling for critical components in WAs (web pages, form elements, hyperlinks and buttons) [13], particularly from the end-user's perspective. IFML is a standard by the Object Management Group (OMG), inspired by the Web Modeling Language (WebML) notation [44]. In a survey conducted by Wright et al. [47], WebML was considered to be the best modeling language for interactive WAs as WebML fulfilled all the criteria required in modeling interactive WAs. WebML performs poorly against some criteria such as supporting the modeling of some elements in interactive WAs (i.e. browser information, user lifecycle or sessions, and the user interface), and limitations in meta-modeling tools and standards. Although Wright et al. propose IAML [45] to overcome WebML's shortcomings, IAML is openly available[2], but the lack of sufficient references causes uncertainties on the kind of learning curve that it presents. On the other hand, IFML has evolved from the ten years of experience with WebML [48], and benefits from the application of WebML in several projects [49, 50]. IFML seems more appropriate based on its ability to model the content, user interaction and control behaviour of the front ends of WAs [4]. IFML does not require a steep learning curve, being an OMG standard that supports visualisation styles that are similar to UML.

III. Pseudo-Genetic Algorithm

In this paper, we define a valid offspring as a test case that is syntactically correct and satisfies the requirements of the SUT when it is executed. An invalid offspring is a test case that is syntactically correct. but is flawed in terms of the requirements of the SUT (e.g., contains a non-existent interaction flow). We present our pseudo-genetic algorithm in Algorithm 1.

Algorithm 1. Pseudo-genetic algorithm

Require: Initial set of test cases *TS*
Require: Maximum number of generation *max*
Require: Global mutation score *gloMS*
Require: General model *GM*

```
1   population ← TS;
2   generation ← 0;
3   gloMS ← (fitness(population));
4   repeat
5   while (generation < max) do
6       P1 ← random(TS);
7       if crossover probability then
8           genes ← random(length(P1));
9           substitutes ← findAlternative(GM, genes);
10          if !isEmpty(substitutes) then
11              selected ← findSubstitute(substitutes, TS);
12              P2 ← random(selected);
13              O1, O2 ← crossover (P1, P2);
14          end if;
15      else
16          gene ← random(length(P1));
17          substitutes ← findAlternative(GM, genes);
18          if isEmpty(substitutes) then
19              selected ← random(substitutes);
20              O1 ← mutate(P1, selected);
21          end if;
22      MS ← (fitness{O1, O2});
23      oldgloMS ← gloMS
24      gloMS ← gloMS ∪ (MS);
25      if (gloMS > oldgloMS) then
26          population ← population ∪{ O1,O2};
27  until gloMS > 80% or maximum resources spent
```

Algorithm 1 is modified from GA. It requires an initial set of test cases *TS*, the general model of the application under study *GM*, and the pre-defined maximum number of generations *max* as input. It optimises the current

[2] Internet Application Modelling Language – http://openiaml.org

population, i.e. the initial set of test cases, to produce an offspring or a pair of offspring test cases, depending on the result of a random selection to perform either a crossover or a mutation. The offspring then joins the current population, and the algorithm repeats itself until the maximum number of generations has been reached.

SBT algorithms require defining the representation of the individuals, and the fitness function that captures the objectives of the search problem and guides the search process [9]. In *MutateIFML*, each test case is a sequence of actions by the user, represented using an IFML model. Each gene represents an IFML element called the *IFMLWindow*.

There are several notable differences between Algorithm 1 and the traditional GA. Firstly, traditional GA consist of a single level cycle of evolution, or optimisation. Our algorithm consists of a two-level cycle of optimisation, with the fitness evaluation phase on a separate level than the rest. During the fitness evaluation, only fit individuals are selected into the population. Secondly, the *GM* is consulted during the recombination and mutation phase. The *GM* was manually created prior to the application of Algorithm 1. We use the *GM* to guide the search process in selecting valid genes when performing crossover or mutation on the parent test cases.

Our algorithm uses the cut and slice crossover technique to perform crossover between two parent test cases on a randomly chosen crossover position, *genes*. Starting with a parent individual, the crossover operator randomly selects the crossover position, *genes* in that parent individual. Next, the algorithm consults the *GM* to find suitable substitutes. If substitutes exist, the algorithm searches the population for individual that possesses identical substitutes. If an individual with the identical substitutes is found in the population, crossover will be performed between that individual and the first parent individual to generate two new individuals. We implement this measure to minimise the possibility of producing invalid offspring.

The mutation operators consist of insert, swap, and delete mutation operators that modify a selected gene of a parent individual. Only one mutation operator is randomly chosen at a time. The insert and swap mutation operators consult the *GM* to retrieve a list of suitable mutant genes. If the list is not empty, the mutation operator will randomly select a mutant gene for the insert or swap operation. As for the delete mutation operator, the general model is used to determine whether the selected gene is appropriate for deletion. If the deletion of the selected gene produces invalid offspring, no offspring will be produced.

IV. MUTATEIFML

MutateIFML works in collaboration with a mutation-testing tool called Humbug and the Selenium WebDriver. *MutateIFML* receives a set of web application test cases and the general model of a web application, both expressed using IFML, and the number of generations as input. Using the pseudo-GA presented in Section III, *MutateIFML* optimises the set of test cases. If *MutateIFML* selects the mutation operators of insert, swap, or delete, it produces one offspring. If *MutateIFML* selects the crossover operator, it produces two offspring. Either way, these offspring are then transformed into an executable Selenium test case. The current *MutateIFML* prototype runs in a given number of generations, and stops when the maximum number of generations is reached.

Prior to running *MutateIFML*, mutants of the SUT were produced separately. These mutants were faulty versions of the SUT. Some were produced by calling Humbug's mutation component, whereas the rest of these mutants were produced by seeding faults into the SUT by hand, based on a fault taxonomy and an online bug reports.

The mutation score of the test case and the mutation score of the population define the fitness function for *MutateIFML*. A test case is deemed fit if it manages to uncover faults in the SUT. The more faults it uncovers, the fitter it becomes in the population.

The mutation scores are calculated at the test case level, *MS*, and the test suite level, *gloMS*. The mutation score *MS* determines the fitness of an individual in the population. This is calculated the percentage of mutants revealed by the individual. The mutation score *gloMS* determines the fitness of the population by calculating the percentage of mutants revealed by the union of all individuals (test cases) in the current population, *TS*.

$$MS = 100 \left(\frac{\text{Number of mutants killed by } T}{\text{Total number of mutants}} \right)$$

$$gloMS = 100 \left(\frac{\text{Number of mutants killed by } TS}{\text{Total number of mutants}} \right)$$

The Selenium offspring test cases are executed to retrieve the fitness of each test case. The execution is achieved by calling Humbug's testing component, which in turn calls PHPUnit to execute Selenium test cases should it finds them in the test folder. *MutateIFML* stops after the specified number of generations is reached.

A. Design

The *MutateIFML* prototype is designed with several features. Its main components are the Humbug controller, the search-based optimisation component, and the Selenium printer. Fig. 1 illustrates *MutateIFML*'s architecture.

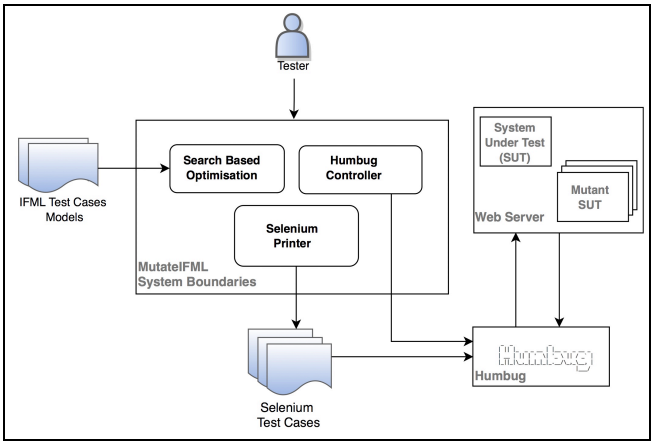

Fig. 1. *MutateIFML* Architecture

B. The Humbug Controller

Humbug is a Hypertext Preprocessor (PHP) mutation-

testing tool [6] that analyses a PHP source file and mutates it to produce a mutant of the file, which is then tested against a test case to determine whether the mutant lives or dies. Humbug has several components, which are used exclusively by *MutateIFML*. The Humbug controller allows *MutateIFML* to select between the mutation component or the testing component of Humbug.

The mutation component analyses the SUT's source files and mutates selected files based on a set of mutation operators predefined in Humbug. This produces several mutants of the SUT. The number of SUT mutants produced depends on the analysis performed by Humbug prior to the mutation of the files.

The testing component is activated by Humbug when it receives a request from *MutateIFML* to do so. Humbug will send a request to its integrated PHPUnit testing framework to execute the test cases on the mutant SUTs that were created. PHPUnit reads the required phpunit.xml configuration file to find the path of the Selenium test cases created by *MutateIFML* from the offspring test cases. This is when the offspring test cases will be tested against the mutated SUTs, and the fitness of each offspring test case will be recorded.

C. The Selenium Printer

To test the offspring created by the optimisation component of *MutateIFML*, the IFML offspring test cases need to be transformed into executable test cases. The Selenium testing framework [51] provides an automated testing tool called Selenium WebDriver that drives the web browser during testing, which requires Selenium test scripts to interact with the SUT.

The Selenium printer component essentially reads the IFML test case's file and transforms certain IFML elements into Selenium commands in Selenese, a HTML-based language. The Selenium printer will first find an *IFMLWindow* element that possesses the default attribute, *isDefault* as the starting point. The default window might contain several input fields, forms, buttons, or links as its child element. The Selenium printer will systematically find and transform the child elements of that default window into Selenium commands. Next, it will find the next window that is connected to the default window. If the next window exists, the transformation process is repeated. The Selenium printer will continue transforming each connected window until it cannot find further connections.

During the transformation process, the Selenium printer also includes test assertion statements. These basic Selenium assertions are added when a certain IFML element is found in the IFML-based test case. Currently, the *VerifyElementPresent*, *VerifyNotText*, and *VerifyText* Selenese commands are used. The Selenium documentation describes the Selenese *VerifyElementPresent* command as a way to "test for the presence of a specific UI element, rather than its content" [52]. On the other hand, the Selenese *VerifyText* and *VerifyNotText* are used to test the presence of IFML's *List* elements, which contain specific texts or messages that are usually triggered by the previous action.

D. The Search-Based Optimisation Componentr

The search-based optimisation component uses the pseudo-GA as its search-based algorithm (see Algorithm 1).

First, the fitness of the individuals in the population is evaluated. This sub-component works closely with Humbug's testing component. Humbug's testing component uses PHPUnit to execute the test cases. We manipulate this feature to our advantage by adding a PHPUnit test script that executes Selenium test cases. Humbug's testing component can record and calculate the test results such as the number of test cases that pass or fail. Some modifications are made to Humbug to produce the test results in a log. The log is then used during the fitness evaluation to calculate the fitness of each individual and the fitness of the population.

Next, the selection of the individual takes place. The selected individuals are then optimised to generate new individuals which are then added to the current population. The fitness of each individual and the current population are again evaluated after the generation of new individuals.

V. Validation Of MutateIFML

We performed the validation of *MutateIFML*'s prototype on a PHP-based web application. *OpenBiblio* is a web-based library management system comprising several library operations such as bibliography cataloguing, circulation, and library member's information management [53]. The validation process is divided into two parts that uses two different source of mutant SUTs. The first group of mutant SUTs was created by Humbug whereas the second group of mutant SUTs was created by hand, which consist of artificial faults created based on the mutation operators proposed by Mansour et al. [54], faults from a bug report of *OpenBiblio* [55], and faults that were found while analysing *OpenBiblio*.

To validate *MutateIFML*, we modelled *OpenBiblio* using the IFML Editor[3]. The modelled WA is used as the *GM* during the validation process. Next, a set of ten IFML test cases was created using the IFML Editor to represent the initial population. These test cases were sequences of user actions on *Login*, *Cataloging*, *Circulation*, *Admin* and *Report* sections of the SUT. The inclusion of various parts of the SUT in modelling the initial population was intentional to minimise the possibility of producing offspring that cluster around certain parts of the SUT only. Valid input values were appended as annotations on the IFML elements that require inputs when executed, so that when these test cases were transformed into Selenium test cases, they would be automatically populated with test data.

We began the validation of *MutateIFML* with 100 generations. Using Humbug's mutation-testing component, 142 mutants of *OpenBiblio* were created. Next, the initial population is tested against these mutant SUTs to obtain the fitness (global mutation score) of the population.

Currently, *MutateIFML* has a predefined maximum number of generations of 100. In each generation, *MutateIFML* randomly selected one individual from the population. Next, either the crossover operator or one of the mutation operators was selected to optimise the individual. If the crossover operator was selected, *MutateIFML* would search for another individual from the population that could produce a pair of valid offspring. If one of the mutation operators (insert, swap, or delete) were selected, *MutateIFML* would produce only one offspring. Then the

[3] Open source IFML editor – http://ifml.github.io

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

offspring was tested against the first group of 142 mutant SUTs of *OpenBiblio*. Later, the offspring was added to the current population, if it managed to kill a mutant SUT that survived the existing testing. *MutateIFML* would reset the optimisation process everytime it found offspring individuals that improves the global mutation score (i.e. kill new mutant SUTs). If this condition was not fulfilled, *MutateIFML* will continue until it reaches the maximum number of generations. We then proceed with the second group of the mutant SUTs. The same validation strategy was applied to 23 hand-seeded mutant SUTs.

VI. Results and Discussion

Fig. 2 presents the result of the validation. The blue line represents the result of the optimisation using Humbug's mutant SUTs (first group), while the red line represents the result of the optimisation using hand-seeded mutant SUTs (second group).

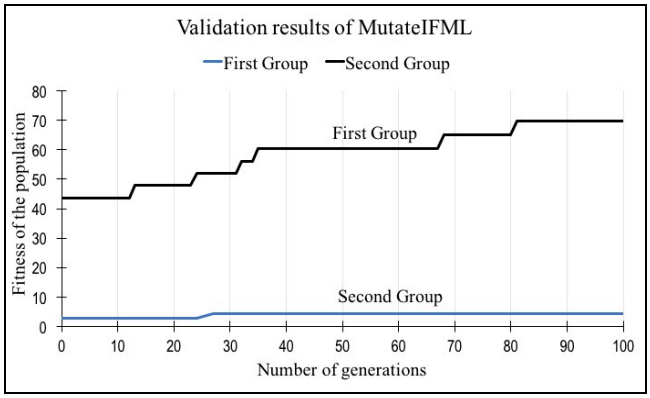

Fig. 2. *MutateIFML*'s result

Using the first group of mutant SUTS, the first test generations recorded 4 out of 142 faulty versions or mutants of *OpenBiblio* killed by the initial population. This result translated to 2.82% global mutation score. After 50 generations, a total of 170 offspring test cases were created. However, only 79 offsprings were found valid after they passed the test against the SUT. When the testing session is performed every iteration, the global mutation score only increased slightly. After 50 iterations, MutateIFML could only produce offsprings that killed two more *OpenBiblio* mutants, with the final global mutation score of 4.23%.

The second group of mutant SUTs performed slightly better, with 10 mutant SUTs killed by the initial population. Therefore, the global mutation score of the initial population is 43.5%. The percentage of mutant SUTs killed continues to grow as the optimisation continues. When the first offspring that killed a mutant SUT was found, the optimisation is reset, with the offspring being included into the initial population. In the end, 115 offspring were produced, with 65 valid offspring when tested against the SUT. The optimisation was iterated 4 times. The final global mutation score was 69.6%, with an additional 6 mutant SUTs killed.

The analysis on the first group of mutant SUTs revealed that out of 142 mutant SUTs of *OpenBiblio*, two mutant SUTs are not created from the files of the SUT, and 87 of them are equivalent mutants. When they were removed, the number of non-equivalent mutant SUTs is reduced to 53. The two non-related mutant SUTs are actually mutations of the PHPUnit *setUp()* file which has to be placed inside the SUT's top directory as a requirement for the PHPUnit's Selenium Extension to run the test. The Humbug test component can avoid specified folders in the SUT during the test, but it cannot avoid files that are placed in the top directory of the SUT.

The non-equivalent mutant SUTs did not seem to produce any differences even after manual testing is performed on them. We realised that the respective mutant SUTs contained changes to the presentation level of the SUT only, thus they did not produce any effect on the SUT when functional testing is performed by Selenium WebDriver.

The analysis also revealed that some mutant SUTs clustered around a few locations in the SUT that were not explored by the test cases. The two most apparent clusters of mutant SUTs are found at the *Offline* and the *Machine-Readable Cataloguing (MARC)* sections of *OpenBiblio*. These two location were not covered by the initial population. The rest of the mutant SUTs modified the links to additional pages when there are more than one pages of search results for the *Cataloguing,* the *Members, or* the *Circulation* tabs. The offsprings never explore the same *IFMLWindow* consecutively. This was a deliberate design decision of the current *MutateIFML* prototype.

The analysis then focused on the second group of mutant SUTs which were created by hand. We discovered a few mutant SUTs that require advanced strategy in terms of test data selection. The test cases contained basic test data that is transformed into Selenium values by the Selenium printer component. Apparently, some remaining mutant SUTs will only be triggered by specific test data. A good example relates to a keyword search performed by *OpenBiblio* in the *Cataloging* tab. The search will produce an error page when a certain keyword, such as "C++" is submitted. Other mutant SUTs were simply not killed due to the lack of coverage from the test cases, since not all forms and links in a visited web page were explored by the offspring. We intend to address the lack of coverage in our future work.

In comparing both groups of mutant SUTs, Humbug produces a lot of equivalent mutants compared to the handmade mutant SUTs. This is attributed to several reasons. Firstly, Humbug is made to create mutations for a PHP software application, but is not specifically for a PHP WA. Secondly, the mutation operators used by Humbug focus on code-level mutation of PHP tokens. Further, Humbug only mutates PHP tokens enclosed within a PHP function. It ignores PHP tokens residing outside a PHP function. As for the handmade mutant SUTs, they are reproduced from three sources selected specifically based on the SUT and its specification. The first source is a set of mutation operators proposed for testing WAs [54]. The second and third sources are real faults discovered from the SUT. Therefore, the handmade mutant SUTs are more reliable and produce fewer mutants that are equivalent. However, Humbug has the advantage of automating the fault-seeding process, whereas the handmade mutant SUTs require significant overhead in compiling faults from the three different sources. Furthermore, since the handmade mutant SUTs are created by the same tester, there is a tendency for the test cases to be biased.

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Even though *MutateIFML* was designed to produce valid offspring, a precautionary measure was implemented to verify whether this design has flaws. Since the general model was created with the assumption that the current version of the SUT is correct, the prototype included the SUT as a temporary oracle to detect the invalid offspring that may have been produced. *MutateIFML* logs all the test results in a text file. If an offspring test case failed the test against the SUT, it would be indicated in the log file, and the failed test case would be treated as an invalid offspring. The invalid offspring would later be analysed to find the design flaws in *MutateIFML* that contributed to the creation of the invalid offspring. For the time being, this oracle is quite useful in helping us to understand the causes of the invalid offspring, and allows targeted debugging to be performed to improve *MutateIFML*.

VII. Future Work

Improvements to *MutateIFML* will be directed towards the search-based optimisation component and the Selenium test case printer component. We propose improvements that will enhance the result of the optimisation process and reduce computation time.

The improvements to optimise components will include the use of the *elite* selection technique used by other SBT approaches [34]. With the *elite* selection technique, the fittest set of test cases will be moved into another population called the *elite* subset. Selection of individuals are then performed on that subset.

With regards to the mutation operators, the current *MutateIFML* moves to the next generation regardless of whether the parent individual and the gene position could produce an offspring. Allowing *MutateIFML* to reselect a different gene and ultimately a different parent in the same generation will indirectly save computation time, since *MutateIFML* does not have to exit the current generation and initiate a new selection phase. Another improvement that will be addressed is incorporating a design solution that addresses the mutation of the same *IFMLWindow* element whenever the user clicks on a self-referencing link or button. This is common in WAs, especially when a web page contains a list of search results that spans over several pages.

Refinement to the fitness evaluation will also be performed. Currently, the fitness evaluation only considers the global mutation score of the population. We plan to add a coverage criterion that not only favours individuals that improve the global mutation score, but also favours individuals that have greater coverage of the SUT.

We discovered that some mutant SUTs created by Humbug were equivalent mutants. The issue of equivalent mutants was limited in the handmade mutant SUTs since they are selected after careful consideration on how they affected the SUT's functionalities. We plan to investigate the handmade mutant SUTs, and find strategies to automate the generation of similar mutant SUTs.

Aside from this, we will investigate on adding more Selenese assertion commands during test case transformation to examine whether more assertions can improve the effectiveness of the offspring test case.

The next case study selected for *MutateIFML* validation is *RosarioSIS*, a school management WA system [56]. *RosarioSIS* is selected based on the active participation of its developers in both updating the WA and recording bugs reported by its users [57]. The validation results from both case studies will then be analysed and the results published.

VIII. Conclusion

We introduced a modified GA called the pseudo-genetic algorithm, which was incorporated into an automated WA testing tool called *MutateIFML*. We described the design of *MutateIFML*. We validated *MutateIFML* on *OpenBiblio*. We found that *MutateIFML* works better with handmade mutant SUTs that were partly replicated from the *OpenBiblio*'s bug report and partly created based on a fault taxonomy for WA. We also identified several limitations of *MutateIFML*, which will be addressed in future work.

References

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[2] M. Harman, "The Current State and Future of Search Based Software Engineering," in *2007 Future of Software Engineering (FOSE)*, pp. 342-357.

[3] G. Fraser, A. Arcuri, and P. McMinn, "A Memetic Algorithm for whole test suite generation," *Journal of Systems and Software,* vol. 103, pp. 311-327, May 2015.

[4] M. Brambilla and P. Fraternali, "Large-scale Model-Driven Engineering of Web User Interaction: The WebML and WebRatio Experience," *Science of Computer Programming,* vol. 89, Part B, pp. 71-87, 1 September 2014.

[5] E. Hieatt and R. Mee, "Going Faster: Testing the Web Application," *IEEE Software,* vol. 19, no. 2, pp. 60-65, Mar/Apr 2002.

[6] P. Brady. (2014, November 23). *Humbug*. Available: https://github.com/padraic/humbug

[7] P. McMinn, "Search-based Software Testing: Past, Present and Future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICST)*, pp. 153-163.

[8] X. S. Yang, *Nature-Inspired Metaheuristic Algorithms*, Frome, United Kingdom: Luniver Press, 2010.

[9] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, "Search Based Software Engineering: Techniques, Taxonomy, Tutorial," in *Empirical Software Engineering and Verification*. vol. 7007, B. Meyer and M. Nordio, Eds. Springer, 2012, pp. 1-59.

[10] W. Miller and D. L. Spooner, "Automatic Generation of Floating-point Test Data," *IEEE Transactions on Software Engineering,* vol. SE-2, no. 3, pp. 223-226, May 1976.

[11] H. Pohlheim, "Genetic and Evolutionary Algorithm Toolbox for Matlab," in *Evolutionäre Algorithmen.* Springer Berlin Heidelberg, 2000, pp. 157-170.

[12] O. Bühler and J. Wegener, "Evolutionary Functional Testing," *Computers & Operations Research,* vol. 35, no. 10, pp. 3144-3160, October 2008.

[13] J. Conallen, "Modeling Web Application Architectures with UML," *Communications of the ACM,* vol. 42, no. 10, pp. 63-70, 1999.

[14] J. Wegener, "Automatic Testing of an Autonomous Parking System using Evolutionary Computation," STZ Sofwaretechnik, SAE 2004 World Congress & Exhibition 2004-01-0459, 2004.

[15] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-based Test Case Generation," *IEEE Transactions on Software Engineering,* vol. 36, no. 6, pp. 742-762, 21 August 2009.

[16] A. Watkins, E. M. Hufnagel, D. Berndt, and L. Johnson, "Using Genetic Algorithms and Decision Tree Induction to Classify Software Failures," *International Journal of Software Engineering and Knowledge Engineering,* vol. 16, no. 02, pp. 269-291, April 2006.

[17] N. Tracey, J. Clark, and K. Mander, "Automated Program Flaw Finding Using Simulated Annealing," in *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 73-81.

[18] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical Combinatorial Interaction Testing: Empirical Findings on Efficiency and Early Fault Detection," *IEEE Transactions on Software Engineering,* vol. 41, no. 9, pp. 901-924, Sepetember 2015.

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

[19] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon, "Genes and Bacteria for Automatic Test Cases Optimization in the .Net Environment," in *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 195-206.
[20] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian, "Heuristics for Fault Diagnosis when Testing from Finite State Machines," *Software Testing, Verification and Reliability,* vol. 17, no. 1, pp. 41-57, March 2007.
[21] B. Chan, J. Denzinger, D. Gates, K. Loose, and J. Buchanan, "Evolutionary Behavior Testing of Commercial Computer Games," in *Congress on Evolutionary Computation, 2004. (CEC)*, pp. 125-132.
[22] T. E. Vos, F. F. Lindlar, B. Wilmes, A. Windisch, A. I. Baars, P. M. Kruse, *et al.*, "Evolutionary Functional Black-Box Testing in an Industrial Setting," *Software Quality Journal,* vol. 21, no. 2, pp. 259-288, June 2013.
[23] A. I. Baars, K. Lakhotia, T. E. Vos, and J. Wegener, "Search-Based Testing, the Underlying Engine of Future Internet Testing," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 917-923.
[24] M. Benedikt, J. Freire, and P. Godefroid. (2002). VeriWeb: Automatically Testing Dynamic Web Sites. in *Proceedings of the 11th International World Wide Web Conference (WWW'2002)* [Online]. Available: https://vgc.poly.edu/~juliana/pub/veriweb-www2002.pdf
[25] H. Reza, K. Ogaard, and A. Malge, "A Model Based Testing Technique to Test Web Applications using Statecharts," in *Fifth International Conference on Information Technology: New Generations (ITNG)*, pp. 183-188.
[26] A. Marchetto and P. Tonella, "Search-Based Testing of Ajax Web Applications," in *1st International Symposium on Search Based Software Engineering (SSBSE)*, pp. 3-12.
[27] G. Fraser and A. Gargantini, "Experiments on the Test Case Length in Specification Based Test Case Generation," in *ICSE Workshop on Automation of Software Test (AST'09)*, pp. 18-26.
[28] G. Fraser and A. Arcuri, "It is Not the Length That Matters, It is How You Control It," in *IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, pp. 150-159.
[29] N. Alshahwan and M. Harman, "Automated Web Application Testing using Search Based Software Engineering," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 3-12.
[30] M. Harman and P. McMinn, "A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, pp. 73-83.
[31] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering,* vol. 16, no. 8, pp. 870-879, 1990.
[32] M. Alshraideh and L. Bottaci, "Search-Based Software Test Data Generation for String Data using Program-Specific Search Operators," *Software Testing, Verification and Reliability,* vol. 16, no. 3, pp. 175-203, September 2006.
[33] R. Zhao, M. R. Lyu, and Y. Min, "Automatic String Test Data Generation for Detecting Domain Errors," *Software Testing, Verification and Reliability,* vol. 20, no. 3, pp. 209-236, September 2010.
[34] F. Bolis, A. Gargantini, M. Guarnieri, and E. Magri, "Evolutionary Testing of PHP Web Applications with WETT," in *Search Based Software Engineering*, G. Fraser and J. T. d. Souza, Eds. Springer, 2012, pp. 285-291.
[35] T. S. Pvt. (2013, November 20). *Sahi*. Available: http://sahi.co.in/
[36] A. Andrews, S. Boukhris, and S. Elakeili, "Fail-safe testing of web applications," in *2014 23rd Australian Software Engineering Conference*, pp. 200-209.
[37] I. K. El-Far and J. A. Whittaker, "Model-Based Software Testing," in *Encyclopedia of Software Engineering*, J. Marciniak, Ed. John Wiley & Sons, Inc., 2001.
[38] J. Ernits, R. Roo, J. Jacky, and M. Veanes, "Model-based Testing of Web Applications using NModel," in *Testing of Software and Communication Systems*. vol. 5826, M. Núñez, P. Baker, and M. G. Merayo, Eds. Springer, 2009, pp. 211-216.
[39] H. Achkar, "Model Based Testing of Web Applications," in *Proceedings of 9th Annual Science Technicians Association of New Zealand Conference (STANZ)*, pp. 1-28.
[40] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing Web Applications by Modeling with FSMs," *Software & Systems Modeling,* vol. 4, no. 3, pp. 326-345, July 2005.
[41] Z. Qian, H. Miao, and H. Zeng, "A Practical Web Testing Model for Web Application Testing," in *Third International IEEE Conference on Signal-Image Technologies and Internet-Based System (SITIS'07)*, pp. 434-441.
[42] J. Offutt and Y. Wu, "Modeling Presentation Layers of Web Applications for Testing," *Software & Systems Modeling,* vol. 9, no. 2, pp. 257-280, April 2010.
[43] F. Ricca and P. Tonella, "Analysis and Testing of Web Applications," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pp. 25-34.
[44] OMG. (2014, January 10). *IFML Specification Beta version (OMG document ptc/2013-03-08)*. Available: http://www.omg.org/spec/IFML/
[45] J. M. Wright, "A Modelling Language for Interactive Web Applications," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 689-692.
[46] N. Koch and A. Kraus. (2002, June). The Expressive Power of UML-Based Web Engineering. in *Second International Workshop on Web-oriented Software Technology (IWWOST02)* [Online]. *16,* 105-120. Available: http://users.dsic.upv.es/~west/iwwost02/papers/koch.pdf
[47] J. Wright and J. Dietrich, "Survey of Existing Languages to Model Interactive Web Applications," in *Proceedings of the Fifth Asia-Pacific Conference on Conceptual Modelling (APCCM)*, pp. 113-123.
[48] WebRatio. (2013, April 18). *OMG adopts the IFML standard, designed by WebRatio*. *News and Events - WebRatio*, [Online]. Available: http://www.webratio.com/site/content/en/news-detail/omg-adopts-ifml-standard-designed-by-webratio
[49] L. Baresi, P. Fraternali, M. Tisi, and S. Morasca, "Towards Model-Driven Testing of a Web Application Generator," in *Web Engineering*. vol. 3579, D. Lowe and M. Gaedke, Eds. Springer, 2005, pp. 75-86.
[50] P. Fraternali and M. Tisi, "Multi-level Tests for Model Driven Web Applications," in *Proceedings of the 10th International Conference on Web Engineering (ICWE)*, pp. 158-172.
[51] OpenQA. (2013, November 23). *Selenium - Web Browser Automation*. Available: http://docs.seleniumhq.org/
[52] OpenQA. (2015, January 5). *Selenium-IDE — Selenium Documentation*. Available: http://www.seleniumhq.org/docs/02_selenium_ide.jsp
[53] B. D. S. Inc. (2013, November 20). *The OpenBiblio Open Source Project on Ohloh*. Available: http://www.ohloh.net/p/openbiblio
[54] N. Mansour and M. Houri, "Testing Web Applications," *Information and Software Technology,* vol. 48, no. 1, pp. 31-42, January 2006.
[55] M. Stetson. (2013, November 20). *mstetson / obiblio - Bitbucket*. Available: https://bitbucket.org/mstetson/obiblio
[56] F. Jacquet. (2016, March 6). *RosarioSIS Student Information System*. Available: https://www.rosariosis.org/
[57] F. Jacquet. (2016, March 6). *RosarioSIS GitHub*. Available: https://github.com/francoisjacquet/rosariosis

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)