Study of development in Search Based Software Testing of Web Applications

Raman Preet Singh, Rupaj Soni and Zachery Thomas

Abstract— **Automated Software Testing has become a very active area of research in recent years. Testing is a widely-used technique for validating Web applications. Automated Software Testing has become a very active area of research in recent years as manual testing tends to become time consuming and error prone. As our reliance on web applications grows, the need to ensure the quality, security and correctness of web applications grows with it. In this paper, we present a systematic literature review describing the changes that have occurred in how we apply automated software engineering and its components to optimize several aspects of web application testing. We see the advances made for search based software optimization in various fields like from dynamic test input generation, reverse engineering of software components, parameter testing to testing an applications non-functional properties.**

KEYWORDS

Search-based software testing; Web applications; Automated test generation; concolic testing

I. INTRODUCTION

The web has had a significant impact on all aspects of our society, from corporate, finance, education, government, entertainment sectors, to our personal lives. The web can be defined as an open source information space where documents and other web resources are identified by URLs, interlinked by hypertext links, and can be accessed via the Internet. There has been tremendous rise in the number of internet users over the past 10 years. Dynamic content web applications, which encompasses almost all the web applications live today, are the most prone to a number of vulnerabilities: SQL Injection, DoS attack, Indirect Object Reference, Cross-site scripting and Cross-site request forgery being among the top 10 vulnerabilities as stated by Open Web Application Security Project(OWASP). Given this rise and the huge attack surface area, the need for highly tested robust applications has risen.

Most of the research work studied in this paper present their own novel techniques to tackle the problem of automated software testing. Some of the papers share common techniques such as concolic testing [9] and some try to address same issue such as of dynamic web application testing [1, 4, 9]. One of the approaches, Search-based software testing, is the area of focus in this paper. In this study, we present a systematic literature review of the web application testing with the focus on use of search based software testing. Next we document our observation in the progress of this field of automated software engineering and how some problems has been tackled while some still remain unsolved. Finally, we present our recommendation and opinion of these works and what we think should be done next in order to make improvements in this domain.

II. MOTIVATION

Manual testing has its limitations and does not always inculcate all the different sides of an application like interoperability, security and dynamics. It is almost obsolete to perform and spend time on any process that can be automated. Computers can be designed to carry out the grunt work while the expensive human labor can be reserved for intelligent design work. This is where the idea of non-domain specific testing framework, which forms the backbone of automated software engineering, comes into play and forms the basis of the papers under review. We know that once a technique or algorithm is developed, it can easily be transformed into domain specific language. In this section, we list down the motivation and hypothesis of the all the papers under consideration.

G. Wassermann et al. [2] made a claim in their paper in 2008, in which they talk about limitations of the static analysis on dynamically generated features which results in high number of bugs that get missed in traditional testing approaches. Therefore, author(s) suggested techniques that can take into account dynamic language features such as scripting language have a behavior that is more string and array centric as opposed to language like Java. And, this required extension to concolic testing approach which author claimed is designed mainly for unit testing. Thus, author(s) also hypothesized to extend concolic testing beyond functions. The authors also realized that there would be a need of a comprehensive testing that takes all of it into account seamlessly.

Web applications content is highly customized and sophisticated and as a result accurate identification of web components becomes highly challenging for testing purposes. Paper by Halfond et. al. [4] aimed at addressing this challenge of interface identification in a dynamic web page. Author(s) proposed to take advantage of symbolic execution to achieve higher precision and accuracy for quality assurance and suggested that this technique can effectively handle the above challenge of interface identification.

Shay Artzy et. al.[5] mentions malformed web pages as a basis for need of testing web pages. The author states that such pages leads to errors at user endpoints. The author of [5] also talks about limitation of the testing and validation tools then that were not ready for the dynamically generated content. Author hypothesized that explicit state model checking for PHP can minimize the size of the failures. They proposed and developed Apollo, a tool to achieve their goal, as they also claimed that testing tools then had very limited functionality.

Another challenging behavior of dynamic web page generation that Halfond et. al. [4] aimed at addressing was of interface identification, due to complex nature of web applications content is highly customized and sophisticated as a result accurate identification of web components becomes highly challenging. Author(s) proposed to take advantage of symbolic execution to achieve higher precision and accuracy for quality assurance and suggested that this technique can effectively handle the above challenge of interface identification.

Nadia alshahwan Et. Al. [1] stresses on the point that non-availability of automated testing leads to hinder continuity and fast availability of these services. They saw a great scope of using Search-based Software Engineering (SBSE) in the field of web application testing. To this effect, they proposed three algorithms, which could be used independently or in combination. In subsequent year Nadia Alshahwan, Mark Harman [6] employed the use of Search-Based Software Engineering techniques such as mutation to design test suits SWAT. The primary motivation here was to incorporate user feedback as a valuable input during automation testing for fault finding. Thus, they propose a methodology Output Uniqueness, based on the difference in elements of the new page compared to the previous, that can complement test suites in testing applications. This technique is further enhanced and presented in [7] where researchers also extend it to achieving high structural whitebox coverage. This extended technique is dependent on the hypothesis that Output Uniqueness exhibits a very high average correlation with statement, branch and path coverage.

Thus, we can see that from 2008 to 2014 different studies are trying to automate the process of testing based on different techniques. The two most popular techniques are Concolic testing and Search-Based Software testing. It is also quite safe to assume that these techniques are somewhat effective and in combination, overlay and solve majority issues leaving some void. However, there is a need for filling in those gaps not through corrective measures but by reaching to the grass root level as to why these gaps exist in the first place. No research has been carried out in this reverse direction of identification of these voids and this is the prime reason behind their existence.

Another noteworthy attribute about these papers is that, although not stated explicitly, none of these testing techniques are single handedly capable of entirely testing an application. This, by definition, is not a real problem if all these authors understand the need for leveraging other testing techniques that would successfully work towards generating a better result.

III. BACKGROUND

A. Web Application Testing

Web application testing is software testing that focuses on web application. Web applications pose a variety of challenges and hence a wide range of test suites are required to deal with different testing aspect such as functional testing, performance testing, browser compatibility, security testing etc.

B. Concolic testing

Concolic testing is a hybrid software verification technique that performs symbolic execution (program variables as symbolic variables) along a concrete execution (testing on particular inputs) path. Symbolic execution is used in conjunction with a constraint solver to generate new test cases.

C. Search Based Software Testing

Search based software engineering (SBSE) is one of the well-known solution to many of software engineerings problems. SBSE proposes that by reformulating a problem as a searchbased optimization problem, an optimized solution to the problem may be found. SBSE converts a software engineering problem into a computational search problem that can be tackled with a metaheuristic. This involves defining a search space, or the set of possible solutions. This space is typically too large to be explored exhaustively, suggesting a metaheuristic approach. The fitness function is then used to measure the quality of potential solutions.

D. Interface identification

It is the technique of identifying the interfaces which are used in a program. Interfaces can be seen as entry points to the application which would spit out dynamic content based on inputs.

E. Penetration testing

It is a process in which the application is tested against malicious attacks in order to reveal security vulnerabilities. A penetration test, is an attack on a computer system that looks for security weaknesses, potentially gaining access to the computer's features and data. The process typically identifies the target systems and a particular goalthen reviews available information and undertakes various means to attain the goal. A penetration test target may be a white box (which provides background and system information) or black box (which provides only basic or no information except the company name). A penetration test can help determine whether a system is vulnerable to attack, if the defenses were sufficient, and which defenses (if any) the test defeated.

F. Explicit-state model checking

It is that type of algorithm that reaches only a single level deep in the application. That means it is an algorithm that always starts the checking again from the same initial state and removes the state reached at the end of each execution.

G. Static analysis

Static program analysis is the analysis of computer software that is performed without actually executing programs. It is the type of debugging technique which is done before the execution of program. It helps to provide a better understanding of the code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding, program comprehension, or code review. Software inspections and software walkthroughs are also used in the latter case. Static analysis tools are a fast and efficient at finding code defects and inconsistencies, especially in large code bases, including older legacy code and newly created code.

H. Genetic algorithm

Genetic algorithms are a way of using the ideas of evolution in computer science. When thinking of the evolution and development of species in nature, in order for the species to survive, it needs to develop to meet the demands of its surroundings. Such evolution is achieved with mutations and crossovers between different chromosomes, i.e., individuals, while the fittest survive and are able to participate in creating the next generation. In computer science, genetic algorithms are used to find a good solution from a very large search space, the goal obviously being that the found solution is as good as possible. To operate with a genetic algorithm, one needs an encoding of the solution, i.e., a representation of the solution in a form that can be interpreted as a chromosome, an initial population, mutation and crossover operators, a fitness function and a selection operator for choosing the survivors for the next generation.

I. Fitness Function

A fitness function is a particular type of objective function that is used to summarize, as a single figure of merit, how close a given design solution is to achieving the set aims. In a search-based approach, the fitness function measures the quality of an individual I within a population P, and is essential to guide the evolution of individuals towards the desired solution.

IV. SAMPLING PROCEDURES

We started with reviewing the works published in 2011 which were related to the domain of automated software engineering. The aim was to select a highly cited paper and then work backwards to recognize the work on which it is built upon and then forwards to understand how the idea proposed by the paper was propagated in subsequent research work. We glanced through various papers manually where we came across a wide range of applications where automated software engineering was used as a technique for various optimization purposes.

We started with Automated Web Application Testing using Search based Software Engineering published in 2011 by Nadia Alshahwan and Mark Harman. We then started to review works published and cited by these authors in their publication backwards in time. Four such works were reviewed. Next, we worked forward in time and reviewed papers that cited the above mentioned publication. This drew a very clear picture of the progress being made in the field and gave us enough tools to evaluate it and make suggestions. This paper presents a high level summary of it all and also some other publications of interest that are related.

V. LITERATURE REVIEW: RETROSPECT

In this section we review the research work done prior to 2011 which gave momentum in the field and led to advances in Automated Software Engineering.

A. Dynamic Test Input Generation for Web Applications[9]

1) Summary: The paper by Wassermann et al [9, 11] addresses differences between typical application (C or Java) testing and web application (PHP or JavaScript) testing. Paper discusses techniques and coding examples for PHP applications. Manual testing requires extensive human effort, which comes at significant cost. Within the domain of automated web application testing, this paper focuses on automatic test case generation, automatic test input generation by modeling string operations, and checking string values against existing policies to prevent SQL injection attacks. They propose using concolic (concrete + symbolic) testing for web applications, and test their methods on three PHP web sites (Mantis, Mambo, and Utopia News) in the context of detecting SQL injections.

2) Approach: Listed below is approach followed by the author -

Constraint generation : The testing framework proposed here represents a symbolic constraint for each functional call that appears in the programs execution.

Resolution of constraints : Finite State Machines (FSMs) were used to invert string operations. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition; this is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

Feedback : Test oracles are required that will give feedback on each execution of the program. Typically this feedback takes the form of pass or fail.

Constraint selection based on relevance : An iterative approach is proposed to narrow the focus of constraint generation to constraints that are relevant to possible failures.

Baseline Results : The authors performed a source to source translation of PHP and wrote down the results in a log file so as to trace the execution. They recorded the results of execution times and corresponding log sizes for Mantis, one of the web applications. They also evaluated how long it took in terms of number of test inputs generated and the total time to generate them. In their observation two applications required relatively few test inputs before they generated an attack.

Table 1 shows how long it took to find an input that caused an SQL Injection attack for each program. The result

TABLE I
ITERATIONS TO FIND AN INJECTION VULNERABILITY

Test Case	Inputs Generated	Time (mm:ss)	Log Size (KB)
Mambo	4	13:02	65
Mantis	5	03:38	19
Utopia	23	05:14	17

is evaluated based on the number of test inputs generated and the total time to generate them.

3) Relevance: This paper applied the symbolic execution to source code when making function calls with SQL queries. Rather than testing the web application completely, it was focused only on testing SQL Injection vulnerabilities. Ultimately, this paper did not contribute much to the web application testing community, seemingly because their results were very niche.

4) Recommendations: a) For conditional expressions in their experiments, their algorithm did not make any approximations by considering only one variable occurrence per expression instance. Thus the conditional expressions on constructed queries in assert statements that had more than one variable occurrence were falsely evaluated.
b) Alternate search based algorithms should be explored further and applied to a larger range of Ajax applications.
c) The Finite State Machine used here can be further improved.

B. Precise Interface Identification to Improve Testing and Analysis of Web Applications[4]

1) Summary: Components of the web application communicate extensively via implicitly defined interfaces to generate dynamic content. The interface identification is the fundamental for many automated quality assurance techniques. But most techniques for identifying web applications are incomplete or imprecise, which hinders the effectiveness of quality assurance techniques. TO deal with this issue, they propose a form of symbolic execution which can abstract away the implementation details and provide an interface for test generation. They test this symbolic executor on several applications, including test-input generation, penetration testing, and invocation verification.

2) Terminology:

- Symbolic execution: A graphical representation of the paths based on which set of inputs trigger parts of a program to execute.
- Domain Constraining Operations: Certain types of operations that we call domain constraining operations, implicitly constrain the domain of an IP. functions that convert an IP value into a numeric value is an example of such an operation.
- Penetration testing: Penetration testing is used to test the immunity of the web application which is exploited by using malicious user input.
- Interface Domain Constraint (IDC): The set of constraints place on an accepted interface along a specific execution path are termed as interface domain constraints (IDC). An accepted interface may have more than one IDC associated with it under certain conditions.

3) Implementation: Their approach can be described to be of these three steps:

Symbolic Transformation : The goal of this step is to transform a web application so that its symbolic execution will provide information about accepted interfaces and interface domain constraints. There are two parts to this transformation. The first is to identify points in the application where symbolic values must be introduced to precisely model the applications IPs. The second is to replace domain-constraining operations with special symbolic operators that will appropriately update the PC as the application is symbolically executed.
Generating Path Conditions : In the second step, they generate the web applications PCs by symbolically executing the transformed web application. Each PC generated in this step corresponds to a family of paths from the entry to the exit of one of the applications web components. The symbolic execution generates the PCs by collecting constraints on the symbolic values during execution of the component. These constraints are created by the execution of operations on symbolic values.
Interface Identification : In the third step, they identify accepted interfaces and IDCs by analyzing the PCs and symbolic states generated in Step 2. To do this, they takes advantage of several insights. First, each IP accessed along a path is added to the symbolic state; therefore, each unique collection of names in ST corresponds to the IP names that define an accepted interface of the component. Second, each constraint in the PC represents part of a domain-constraint on the value of a particular IP; therefore, the PC corresponds to an IDC. Third, the IDC corresponds to the accepted interface identified in the symbolic state.

4) Evaluation: The authors developed a prototype tool called WAM-SE (Web Application Modeling with Symbolic Execution) to evaluate their experiment. The implementation consists of three modules: transform, SE engine, and PC analysis, which correspond to the three steps of their implementation.

Transform : The input to this module is the byte code of the web application and the specification of program entities are considered to be symbolic .

se engine : The se engine module implements the symbolic execution described as a part of generation of path conditions. The input to this module is the bytecode of the transformed web application, and the output is the set of all PCs and corresponding symbolic states for each component in the application.

PC Analysis : The input to this module is the set of PCs and symbolic states for each component in the application, and the output is the set of Interface Domain Constraints and accepted interfaces. The module iterate over every PC and symbolic state, identifies the accepted interfaces, and associates the constraints on each IP with its corresponding accepted interface.

Subject	Vulnerable Parameters			
	W_{df}	W_{se}	*Spi.*	DFW
Bookstore	11	36	7	5
Classifieds	14	31	4	18
Employee Dir.	11	19	1	4
Events	11	23	4	2
Total	47	109	16	29

TABLE II
RESULTS OF PENETRATION TESTING

Subject	# Size of test suite			
	W_{df}	W_{se}	*Spi.*	DFW
Bookstore	258,565	10,634	68,304	33,279
Classifieds	47,352	3,968	7,238	10,732
Employee Dir.	627,820	3,772	46,099	54,887
Events	36,448	1,735	4,145	5,566
Average	242,546	5,027	31,447	26,116

TABLE III
TEST-INPUT GENERATION AND COVERAGE

5) *Results:*

6) *Relevance:* Halfond et al's approach was limited to Java applications, whereas Alshahwan et al [1] used the same approach but extended the same idea further to PHP applications. Also, Alshahwan et al [1] performed a similar static analysis over the GET, POST, and REQUEST statements for handling interface determination. Most importantly, they also did a good job of handling ill-defined input with an abstract layer. Thus, this work proved to be highly relevant to the works in future and marked significant contributions to testing of web applications.

C. *Finding Bugs in Dynamic Web Applications [5]*

1) *Summary:* The authors devised their own testing oracle Apollo to determine if the output of a dynamic web applications, is syntactically correct along with automating the process of sorting and minimizing the inputs that leads to faults. Apollo consists of three major components, Executor, Input Generator, and Bug Finder as shown in Figure 1 below along with many sub components.

Apollo monitors the program to record the dependence of control-flow on input. Additionally, for each execution Apollo determines whether execution failures or HTML failures occur (for HTML failures, an HTML validator is used as an oracle). Apollo automatically and iteratively creates new inputs using the recorded dependence to create inputs that exercise different control flow.

2) *Keywords:*

- Dynamic test generation : Dynamic testing measures how accurately software responds to input, and how difficult it is to generate sufficient tests (especially for web pages).
- Program testing : Performing an investigation to gauge the quality of a piece of software, especially when used under specific conditions.
- Program verification : Program verification checks that the program was written correctly, and is typically associated with static testing.

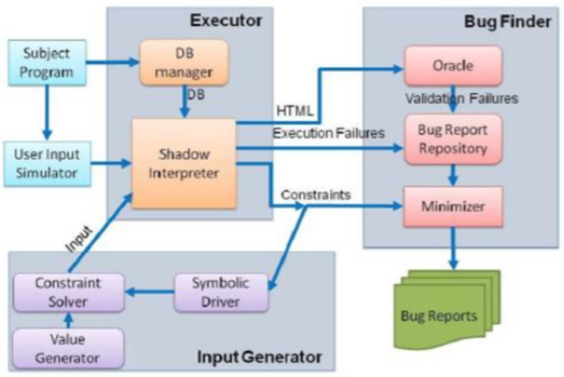


Fig. 1. Architecture of Apollo

3) *Relevance:* This paper laid down the foundations of the work of Alshahwan et al. The former paper deals with optimization for statement coverage while the latter optimize the branch coverage. The authors have tackled the problem by explicit state model checking which tries every possible combination which is a highly complex scenario. Alshahwan et al has taken a smarter approach so as to reduce run time without reducing too much solution quality.

VI. LITERATURE REVIEW: FUTURE

A. *Automated Web Application Testing Using Search Based Software Engineering[1]*

1) *Summary:* The main idea of this paper is how to improve automated web application testing by increasing branch coverage with reduction in testing effort. This paper introduces three algorithms and a tool called "SWAT" and explains the positive impact of these algorithms in efficiency and effectiveness of traditional search based techniques exploiting both static and dynamic analysis. Each improvement is separately evaluated in an empirical study on 6 real world web applications.

2) *Delivery Tool:* The algorithm starts with a static analysis phase that collects static information to aid the subsequent search based phase. The search based phase uses an algorithm that is derived from Korels Alternating Variable Method (AVM) but which additionally incorporates constant seeding and Dynamically Mined Values (DMV) from the execution and web pages constructed by the application as it executes.

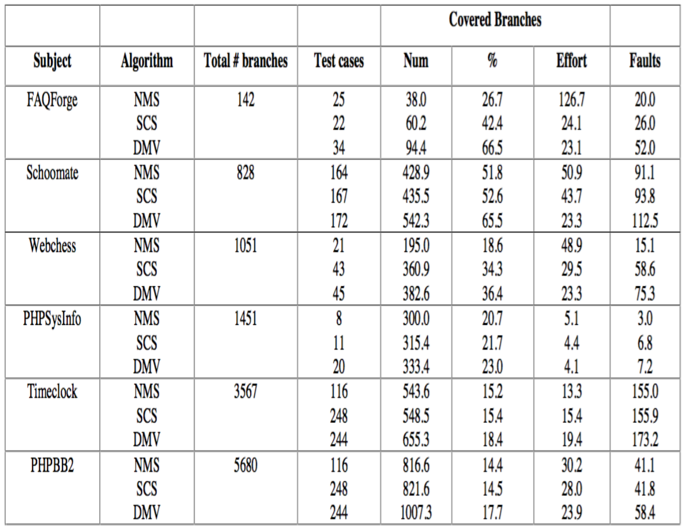
SWAT consists of two major tools: 1. Search based Tester and 2.Test Harness. The Search based Tester uses the transformed source code and the analysis data to implement the input generation described in three different Algorithms. The

Test Harness uses the generated test data to run the tests on the original source code and to produce coverage and bug data.

3) Evaluation: The authors designed their experiments to provide answer to the following set of questions:
1. How does each of the enhancements affect branch coverage?
2. How does each of the enhancements affect efficiency of the approach?
3. How does each of the enhancements affect fault finding ability?

To measure efficiency, they recorded average execution times and measured effort for each approach over 30 runs.

TABLE IV
AVERAGE COVERAGE AND EXECUTION TIME RESULTS OBTAINED BY RUNNING EACH ALGORITHM 30 TIMES FOR EACH APPLICATION WITH THE SAME BUDGET OF EVALUATIONS PER BRANCH FOR EACH VERSION



				Covered Branches			
Subject	Algorithm	Total # branches	Test cases	Num	%	Effort	Faults
FAQForge	NMS	142	25	38.0	26.7	126.7	20.0
	SCS		22	60.2	42.4	24.1	26.0
	DMV		34	94.4	66.5	23.1	52.0
Schoomate	NMS	828	164	428.9	51.8	50.9	91.1
	SCS		167	435.5	52.6	43.7	93.8
	DMV		172	542.3	65.5	23.3	112.5
Webchess	NMS	1051	21	195.0	18.6	48.9	15.1
	SCS		43	360.9	34.3	29.5	58.6
	DMV		45	382.6	36.4	23.3	75.3
PHPSysInfo	NMS	1451	8	300.0	20.7	5.1	3.0
	SCS		11	315.4	21.7	4.4	6.8
	DMV		20	333.4	23.0	4.1	7.2
Timeclock	NMS	3567	116	543.6	15.2	13.3	155.0
	SCS		248	548.5	15.4	15.4	155.9
	DMV		244	655.3	18.4	19.4	173.2
PHPBB2	NMS	5680	116	816.6	14.4	30.2	41.1
	SCS		248	821.6	14.5	28.0	41.8
	DMV		244	1007.3	17.7	23.9	58.4

Table II summarizes the results obtained by the experiment. Coverage results are reported together with the number of test cases generated to achieve this coverage. In Table II crashes indicate PHP interpreter fatal execution errors; these are errors that cause the execution of scripts to abort. Errors indicate PHP interpreter warnings; these are defined in the PHP manual as non-fatal errors

4) Areas of Improvement: 1. The test is set up on a simple hardware machine and is run locally. Thus it fails to take into consideration the latency as imbibed as being a part of the network. It would have been more value added if they had tested by setting up remote services in a distributed environment.
2. The only criteria mentioned for selecting the six web applications is size. There are many other factors which should have been considered while making this choice.
3. All the applications chosen for the study are PHP applications. To cover major areas of web applications they could have selected other applications implemented by other languages like Java to see the overall effectiveness and efficiency of this approach.

B. The Seed is Strong: Seeding Strategies in Search-Based Software Testing [12]

1) Summary: This paper considers the aspect of the initial population of the search, and presents and studies a series of techniques to improve search-based test data generation for object oriented software. New and typical strategies to seed the initial population as well as to seed values introduced during the search when generating tests for object-oriented code were evaluated. The ultimate goal is to reduce the number of test cases generated that maximize the branch coverage and at the same time have as low secondary objectives as possible, as these need manual verification.

2) Delivery tool: The authors of this paper have EVOSUITE, an advanced tool based on Genetic Algorithm, to derive test suites for classes. EVOSUITE features the novel approach of evolving whole test suites at the same time, and the authors felt that there are plenty of opportunities to improve search operators such as crossover. The EVOSUITE tool operates on Bytecode. Compilation produces a binary representation of the classes that is close to machine code, yet retains parts of the original structure(classes and methods). The performance of EVOSUITE at different search budget intervals for a specific class are shown in Figure 2. In particular, four different seeding strategies with the baseline being- no seeding. Figure 2 clearly shows that the differences in performance of the seeding strategies are more marked for low budgets, whereas they nearly disappear for higher search budgets.

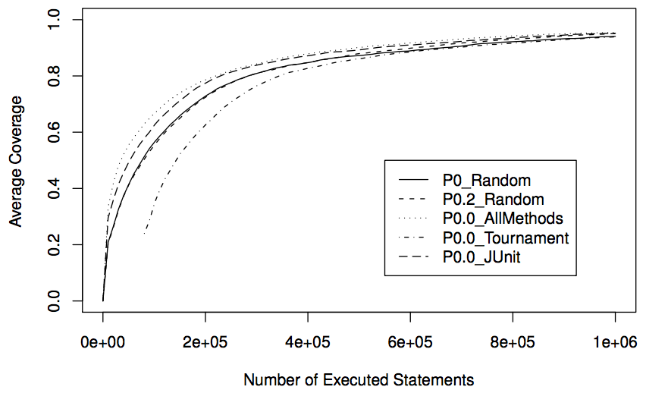


Fig. 2. Coverage of the best individual during the search: Earlier during the search the difference between the seeding strategies is more significant, while at later points all strategies converge. P0 Random denotes random initial tests with no constant seeding, P0.2 Random uses constants with probability 0.2, etc.

3) Related Work: The authors discussed a few papers on seeding strategies to improve the search. They proved to be enormous help to further increase the result based on Search based testing. Some of the papers/work they mentioned were: a) Langdon and Nordin [13] studied a seeding strategy

in order to improve the ability of a classifier/repressor to generalize.
b) White et al. [14] studied several different seeding strategies to initialize a Genetic Programming population for optimizing execution time of a given input program.
c) In the context of testing real-time systems to find worst case execution times, Tlili et al. [15] applied seeding strategies as well. Given the execution time of the system under test as the fitness function to optimize, instead of starting from scratch, they used a test case with high coverage as seed to start the search from.
d) Miraz et al. created the initial population by selecting the best individuals out of a larger pool of randomly generated individuals.[16]

4) Results: The seeding techniques were evaluated with a large case study, comprising 20 projects for a total of 1,752 public classes and 85,503 bytecode level branches, and statistical procedures were followed to assess the scientific validity of results. The results show that, with high statistical confidence, seeding strategies do improve the performance of the employed SBST technique.

Project	AllMethods				Tournament			
	Coverage	$\hat{A}_{12}$	$\% > 0.5$	$\% < 0.5$	Coverage	$\hat{A}_{12}$	$\% > 0.5$	$\% < 0.5$
COL	0.73	0.492	0.036	0.051	0.73	0.491	0.022	0.109
CCL	0.87	0.503	0.071	0.000	0.87	0.500	0.000	0.000
CCD	0.88	0.503	0.048	0.000	0.87	0.495	0.000	0.048
CCO	0.91	0.502	0.041	0.004	0.91	0.498	0.004	0.016
CMA	0.78	**0.521**	0.182	0.008	0.73	**0.482**	0.016	0.162
CPR	0.93	0.498	0.005	0.024	0.93	0.499	0.005	0.010
GCO	0.75	0.511	0.110	0.011	0.74	0.500	0.044	0.033
ICS	0.85	0.501	0.000	0.000	0.85	0.498	0.000	0.000
JCO	0.82	0.500	0.033	0.000	0.81	0.495	0.000	0.000
JDO	0.73	0.503	0.053	0.018	0.72	0.496	0.035	0.000
JGR	0.75	0.499	0.007	0.007	0.75	0.500	0.007	0.000
JTI	0.85	**0.513**	0.183	0.023	0.84	0.496	0.023	0.069
NXM	0.61	0.627	0.000	0.000	0.59	0.569	0.000	0.000
NCS	0.97	0.500	0.000	0.000	0.97	0.499	0.000	0.091
REG	0.75	0.501	0.333	0.333	0.74	0.487	0.000	0.000
SCS	0.63	0.501	0.000	0.000	0.64	0.504	0.000	0.000
TRO	0.88	0.500	0.010	0.010	0.88	0.499	0.010	0.019
XEN	0.65	0.502	0.000	0.000	0.65	0.499	0.000	0.000
XOM	0.76	0.499	0.000	0.024	0.76	0.499	0.000	0.012
ZIP	0.80	0.503	0.000	0.000	0.81	0.515	0.000	0.000
Average	0.83	0.509	0.056	0.026	0.82	0.501	0.008	0.028

TABLE V
FOR EACH PROJECT, AVERAGE COVERAGE ON ALL OF ITS CLASSES WHEN ALLMETHODS AND TOURNAMENT SEEDING STRATEGIES ARE EMPLOYED. THE A12 ARE IN RESPECT TO THE DEFAULT RANDOM STRATEGY, AND ARE CALCULATED BY AGGREGATING ALL RUNS OF ALL CLASSES PER PROJECT (IN BOLD IF STATISTICALLY SIGNIFICANT). ON HIGHER GRANULARITY, IT IS REPORTED THE PERCENTAGE OF CLASSES FOR WHICH WE HAVE A SIGNIFICANT A12 ¿ 0.5 AND A12 ¡ 0.5.

C. Genetic programming for reverse engineering

1) Summary: This paper presents a summary of applications of SBSE to reverse engineering. This paper focuses on reverse engineering and the considerable potential for the development of new forms of Genetic Programming (GP) and Genetic Improvement (GI) to reverse engineering. This includes work on SBSE for remodularisation, refactoring, regression testing, syntax-preserving slicing and dependence analysis, concept assignment and feature location, bug fixing, and code migration.

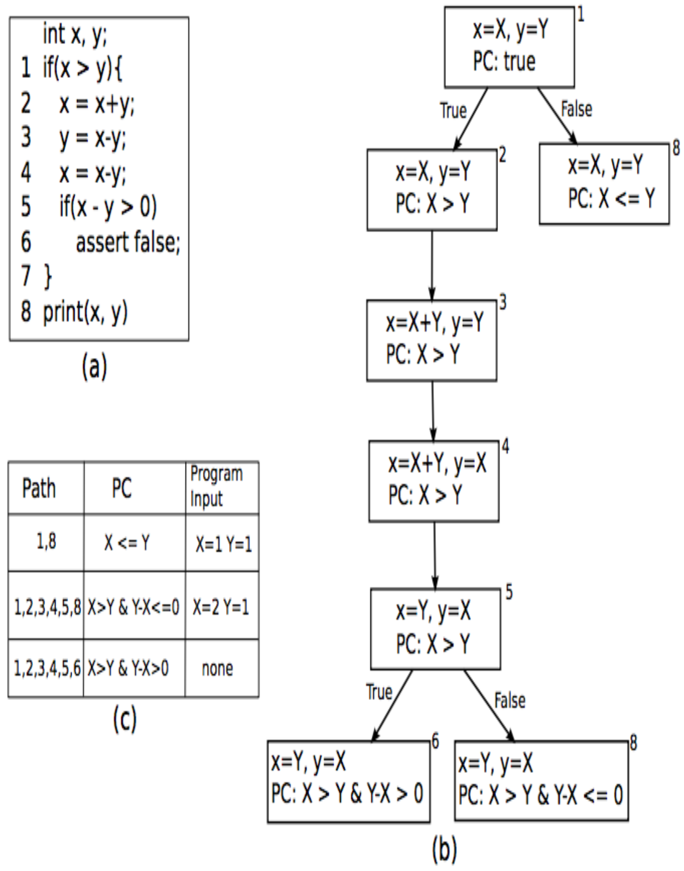
2) Implementation: At a high level, key steps likely to occur in a software transplant algorithm to add feature F from source System D (the donor) to destination System H (the host):
1) Localize: Identify and localize the code that implements F (this might use, for example, concept and feature location.
2) Abstract: Construct an abstraction AF from DF, retaining control and data flow directly related to F in the donor but abstracting references to D-specific identifiers so that these become parameterized.
3) Target: Find locations HF in the host, H, where code implementing F could be located.
4) Interface: Construct an interface, I and add it to the host, H, allowing the resulting combination H [I] to act as a harness into which candidate transplants can be inserted and evaluated.
5) Insert: Instantiate and concretize a candidate transplant (concretized from AF) at HF.

3) Areas of Improvement: 1. The results have not been presented comprehensively so that they can reviewed and conclusions can be drawn from them.
2. No mention of how reverse engineering can be further studied and applied in the field of Search based Software testing.

D. An orchestrated survey of methodologies for automated software test case generation [8]

1) Summary: Anand et al presented an orchestrated survey of the most prominent techniques for automatic generation of software test cases, reviewed in self-standing sections. The techniques presented include:
(a) structural testing using symbolic execution, (b) model-based testing, (c) combinatorial testing, (d) random testing and its variant of adaptive random testing, and (e) search-based testing.

In contrast to black box test data generation approaches, which generate test data for a program without considering the program itself, white box approaches analyze a programs source or binary code to generate test data. One such white box approach, which has received much attention from researchers in recent years, uses a program analysis technique called symbolic execution. Symbolic execution uses symbolic values, instead of concrete values, as program inputs, and represents the values of program variables as symbolic expressions of those inputs. At any point during symbolic execution, the state of a symbolically executed program includes the symbolic values of program variables at that point, a path constraint on the symbolic values to reach that point, and a program counter.



```
  int x, y;
1 if(x > y){
2   x = x+y;
3   y = x-y;
4   x = x-y;
5   if(x - y > 0)
6     assert false;
7 }
8 print(x, y)
```

Path	PC	Program Input
1,8	X <= Y	X=1 Y=1
1,2,3,4,5,8	X>Y & Y-X<=0	X=2 Y=1
1,2,3,4,5,6	X>Y & Y-X>0	none

Fig. 3. (a) Code that swaps two integers, (b) the corresponding symbolic execution tree, and (c) test data and path constraints corresponding to different program paths.

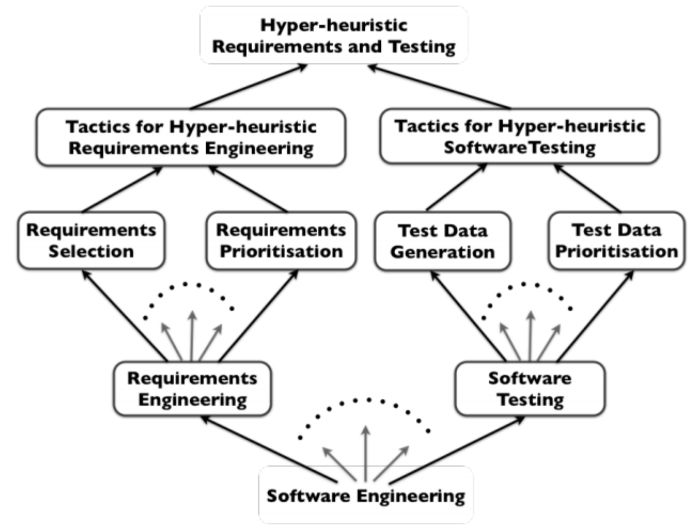
2) Hypothesis: The authors have presented an answer to the most important question in the research agenda of SBSE: *Can we radically increase automation by integrating SBST with other forms of SBSE?*

The authors suggests that although more work can be done in the field, such approaches, on their own, can only offer optimization of a narrow set of very similar engineering activities. That is we will have optimized test input generation and (separately) optimized requirements prioritization. SBST needs to make a transition from solving instances to automatically finding tactics that solve instances. This will increase the abstraction level at which we apply SBSE, as indicated in the upper tree in Figure 4, drawing together sets of related software engineering activities.

3) Relevance: This paper indicates that ”web application testing” is an area where SBSE has made substantial foothold. Another noteworthy contribution is the emphasis made by Afzal et al which concentrated on the trends in metaheuristic usage over time and the clusters of researchers working with these metaheuristics.

VII. Commentary

This report is a summarized form of particular results of search-based techniques applied to automated tests for



Hyper-heuristic Requirements and Testing
Tactics for Hyper-heuristic Requirements Engineering
Tactics for Hyper-heuristic SoftwareTesting
Requirements Selection
Requirements Prioritisation
Test Data Generation
Test Data Prioritisation
Requirements Engineering
Software Testing
Software Engineering

Fig. 4. Hyper-Heuristic SBSE: using Hyper Heuristics, we may be able to develop tactics an strategies that will unite different software engineering activities with SBST.

web applications. The initial papers have presented some novel solutions which have laid down groundwork for the future studies in the field which have later been tweaked and improved. It is quite safe to say that web application testing has become reliable to a certain extent.

The following areas are considered to be more promising and are potential topics for future work in Automated Web Application Testing in SBSE: 1. With the increasing complexity of software systems, the feature models will grow exponentially, which in turn will lead to an exorbitant amount of product branches. Using SBSE developed techniques to control branchomania by identifying branch similarities, extracting parameters and subsequent searching for suitable tunings yield individual products. A set of child branches of a shared parent could thereby be merged into a single modified parent with an additional set of parameters that capture the variability previously present in the children. In this way, a combination of parameter extraction and tuned parameter instantiation could merge several products into a single parameterized product.
2. To exploit the current improvements in hardware specifically multicore processors, search based software engineers have realized the possibility of parallelization as a route to scalability using computing clusters. Parallel SBSE may prove to be particularly important in scaling computational search algorithms to handle large-scale software product lines.
3. Symbolic execution differs other techniques for automatic test-generation in its use of program analysis and constraint solvers. However, it can be used in combination with those other techniques (e.g., search-based testing). An extensive body of prior research has demonstrated the benefits of symbolic execution in automatic test-data generation. More research is needed to improve the techniques usefulness on real-world programs. The fundamental problems that

the technique suffers from are long-standing open problem. Thus, future research, in addition to devising more effective general solutions for these problems, should also leverage domain-specific (e.g., testing of smart-phone software) or problem-specific (e.g., test-data generation for fault localization) knowledge to alleviate these problems.
4. Combinatorial testing is a promising area of research for automated software test generation, with opportunities to enhance new domains of its application. Fruitful future research directions for generating CIT samples could include automated model extraction, adapting to model evolution, and developing techniques that re-use or share information between different test runs. In addition to applying CIT to novel application domains, an area of potential for improvement in this direction is the combination of program analysis techniques with CIT to refine the sample space, and to target specific interactions at the code (as opposed to only the specification) level.
5. A six PHP web application dataset was used in two papers, and a three PHP web application dataset was used in another paper with an emphasis on SQL injections. The PHP datasets used in the papers here were small and not very regular. The researchers should explore more varying datasets such as Java, Android etc to test the robustness and scalability of the Search based software techniques been used.

VIII. Conclusion

We have reviewed 8 years of work done in the field of web application testing using Search Based Software Engineering. This field has seen some tremendous growth in techniques such as test case generation, interface implementation, Concolic testing and reverse engineering.

We found that SWAT algorithms significantly enhance the efficiency and effectiveness of traditional search based techniques. These algorithms performed both static and dynamic analysis which improved branch coverage and reduced test effort. Also we found that using symbolic execution techniques led to improvements in several important quality assurance techniques for web applications: test-input generation, penetration testing, and invocation verification.

The testing performed in this paper was restricted to a specific language such as PHP or Java. From our understanding, testing will always need to evolve as new languages keep on evolving which is a challenge. The effective solution to this problem is usage of combination of multiple techniques as none is complete in itself.

Finally, the advent of automated web application testing has significantly reduced the time and effort spent in testing. These solutions could be made even better after feedback from the industry given the fact that these tests are still not so prevalent in the industry.

Acknowledgment

We would like to thank Prof. Tim Menzies and teaching assistant George Mathew for their continuous support.

References

[1] Alshahwan N., and Harman M. Automated web application testing using search based software engineering. In Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on (Nov 2011), pp. 3-12.

[2] Afza, W., Torkar R., and Feldt R. A systematic review of search-based testing for non-functional system properties. Inf. Softw. Technol. 51, 6 (June 2009), 957-976.

[3] Alshraideh, M., and Bottaci, L. Search-based software test data generation for string data using program-specific search operators: Research articles. Softw. Test. Verif. Reliab. 16, 3 (Sept. 2006), 175-203.

[4] William G.J. Halfond, Saswat Anand, and Alessandro Orso. 2009. Precise interface identification to improve testing and analysis of web applications. Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09). ACM, New York, NY, USA.

[5] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit state model checking. IEEE Transactions on Software Engineering, 36:474-494, 2010.

[6] Nadia Alshahwan , Mark Harman, Augmenting test suites effectiveness by increasing output diversity, Proceedings of the 34th International Conference on Software Engineering, June 02-09, 2012, Zurich, Switzerland.

[7] Krutz D., Mirakhorli M., Malachowsky S., Ruiz A., Peterson J., Filipski A., and Smith J. A dataset of open-source android applications. In Proceedings of the 12th Working Conference on Mining Software Repositories (Piscataway, NJ, USA, 2015), MSR '15, IEEE Press, pp. 522-525.

[8] Anand, Saswat, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. "An orchestrated survey of methodologies for automated software test case generation." Journal of Systems and Software 86, no. 8 (2013): 1978-2001.

[9] Wassermann G., Yu D., Chander A., Dhurjati D., Inamura H., & Su, Z. (2008, July). Dynamic test input generation for web applications. In Proceedings of the 2008 international symposium on Software testing and analysis (pp. 249-260). ACM.

[10] Nadia Alshahwan , Mark Harman, Coverage and fault detection of the output-uniqueness test selection criteria, Proceedings of the 2014 International Symposium on Software Testing and Analysis, July 21-25, 2014, San Jose, CA, USA.

[11] Wassermann G., and Su Z. Sound and precise analysis of web applications for injection vulnerabilities. SIGPLAN Not. 42, 6 (June 2007), 3241.

[12] Fraser, Gordon, and Andrea Arcuri. "The seed is strong: Seeding strategies in search-based software testing." Software Testing, Verification and Validation (ICST), 2012 IEEE.

[13] W. B. Langdon and P. Nordin, Seeding genetic programming populations, in Proceedings of the European Conference on Genetic Programming (EuroGP), 2000, pp. 304315.

[14] D. White, A. Arcuri, and J. Clark, Evolutionary improvement of programs, IEEE Transactions on Evolutionary Computation (TEC), vol. 15, no. 4, pp. 515538, 2011.

[15] M. Tlili, S. Wappler, and H. Sthamer, Improving evolutionary real-time testing, in Genetic and Evolutionary Computation Conference (GECCO), 2006, pp. 19171924.

[16] M. Miraz, P. Lanzi, and L. Baresi, Improving evolutionary testing by means of efficiency enhancement techniques, in IEEE Congress on Evolutionary Computation (CEC), 2010, pp. 18.

[17] Harman, Mark, William B. Langdon, and Westley Weimer. "Genetic programming for reverse engineering." Reverse Engineering (WCRE), 2013 20th Working Conference on. IEEE, 2013.

[18] Hamasaki, K., Kula, R., Yoshida, N., Cruz, A., Fujiwara, K., and Iida, H. Who does what during a code review? datasets of oss peer review repositories. In Proceedings of the 10th Working Conference on Mining Software Repositories (Piscataway, NJ, USA, 2013), MSR 13, IEEE Press, pp. 4952.

[19] Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., and Su, Z. Dynamic test input generation for web applications. In Proceedings of the 2008 International Symposium on Software Testing and Analysis (New York, NY, USA, 2008), ISSTA 08, ACM, pp. 249260.

[20] Serdar Doan, Aysu Betin-Can, Vahid Garousi, Web application testing: A systematic literature review, Journal of Systems and Software, 91, p.174-201, May, 2014B. Korel. Automated software test data generation. IEEE Transactions on Software Engineering, 16(8):870879, 1990.