Information Systems 43 (2014) 20–54

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Two decades of Web application testing—A survey of recent advances

Yuan-Fang Li *, Paramjit K. Das, David L. Dowe

Clayton School of Information Technology, Faculty of Information Technology, Monash University, Clayton, VIC 3800, Australia

ARTICLE INFO

Article history:
Received 1 February 2013
Received in revised form
21 December 2013
Accepted 1 February 2014
Recommended by: F. Carino Jr.
Available online 14 February 2014

Keywords:
Software testing
Web applications
World Wide Web
Web testing
Survey

ABSTRACT

Since its inception of just over two decades ago, the World Wide Web has become a truly ubiquitous and transformative force in our life, with millions of Web applications serving billions of Web pages daily. Through a number of evolutions, Web applications have become interactive, dynamic and asynchronous. The Web's ubiquity and our reliance on it have made it imperative to ensure the quality, security and correctness of Web applications. Testing is a widely used technique for validating Web applications. It is also a long-standing, active and diverse research area. In this paper, we present a broad survey of recent Web testing advances and discuss their goals, targets, techniques employed, inputs/outputs and stopping criteria.

Contents

* Corresponding author.
E-mail addresses: yuanfang.li@monash.edu (Y.-F. Li), paramjit.das@monash.edu (P.K. Das), david.dowe@monash.edu (D.L. Dowe).

http://dx.doi.org/10.1016/j.is.2014.02.001
0306-4379

1. Introduction

Software testing [1] has been widely used in the industry as a quality assurance technique for the various artifacts in a software project, including the specification, the design, and source code. As software becomes more important and complex, defects in software can have a significant impact to users and vendors. Therefore, the importance of planning, especially planning through testing, cannot be underestimated [1]. In fact, software testing is such a critical part of the entire process of producing high-quality software that an industry may devote as much as 40% of its time on testing to assure the quality of the software produced.

In software testing, a suite of test cases is designed to test the overall functionality of the software—whether it conforms to the specification document or exposes faults in the software (e.g., functionality or security faults). However, contrary to the preconceived notion that software testing is used to demonstrate the absence of errors, testing is usually the process of finding as many errors as possible and thus improving assurance of the reliability and the quality of the software [1]. This is because, in order to demonstrate the *absence* of errors in software, we would have to test for all possible permutations for a given set of inputs. However, realistically, it is not possible to test for all the permutations of a given set of input(s) for a given program, even for a trivial program. For any non-trivial software systems, such an exhaustive testing approach is essentially technologically and economically infeasible [1]. The main objectives of any testing technique (or test suite) can be summarised as:

- Testing is carried out mainly to demonstrate the presence of errors that exist during a program execution.
- A good testing technique will have a higher chance of discovering an error.
- A successful test case should discover a new fault or a regression fault.

Ever since the creation of the World Wide Web in the early 1990s [2], there has been a tremendous increase in the usage of Web applications in our daily lives. The idea behind the World Wide Web was possibly envisioned by C.S. Wallace as early as 1966 [3, pp. 244–245], where he envisioned that a central computing system (or the server), or a bank of computers, could be used to carry out various computing tasks, such as paying bills, ordering goods, carrying out engineering tasks, etc., for a large number of users. In these instances, the time required would be shared equally amongst all users, which would make the process economically feasible. This concept was appropriately labelled "Time-Sharing", since the time would be shared amongst all users.

A *Web application* is a system which typically is composed of a database (or the back-end) and Web pages (the front-end), with which users interact over a network using a browser. A Web application can be of two types – *static*, in which the contents of the Web page do not change regardless of the user input; and *dynamic*, in which the contents of the Web page may change depending on the user input, user interactions, sequences of user interactions, etc.

The profound transformative impact the Web and Web applications have brought about on our society has long been acknowledged. Somewhat surprisingly, however, there seems to be very limited research that has been done in surveying the different recent advancements made in the field of Web application testing over the past 20 years. To the best of our knowledge, the only other surveys in this field consists of an early review by Di Lucca and Fasolino [4] on general approaches to Web application testing, and a survey by Alalfi et al. [5] focussed on the modelling aspects of Web application testing. Therefore, this survey paper provides a much needed source of detailed information on the progress made in and the current state of Web application testing.

Compared to traditional desktop applications, Web applications are unique in a number of ways, and such uniqueness presents new challenges for their quality assurance and testing.

- Firstly, Web applications are typically multilingual. A Web application usually consists of a server-side backend and a client-facing frontend, and these two components are usually implemented in different programming languages. Moreover, the frontend is also typically implemented with a

mix of markup, presentation and programming languages such as HTML, Cascading Style Sheets (CSS) and JavaScript. The presence of multiple languages in a Web application poses additional challenges for fully automated *continuous integration* (CI) practices, as test drivers for different languages need to be integrated into the CI process and managed coherently.

- Secondly, the operating environment of typical Web applications is much more open than that of a desktop application. Such a wide visibility makes such applications susceptible to various attacks, such as the distributed denial-of-service (DDOS) attacks. Moreover, the open environment makes it more difficult to predict and simulate realistic workload. Levels of standards compliance and differences in implementation also add to the complexity of delivering coherent user experiences across browsers.
- Thirdly, a desktop application is usually used by a single user at a time, whereas a Web application typically supports multiple users. The effective management of resources (HTTP connections, database connections, files, threads, etc.) is crucial to the security, scalability, usability and functionality of a Web application. The multi-threaded nature of Web applications also makes it more difficult to detect and reproduce resource contention issues.
- Last but not least, a multitude of Web application development technologies and frameworks are being proposed, actively maintained and fast evolving. Such constant evolution requires testing techniques to stay current.

The rest of this paper is organised as follows. We begin in Section 2 by motivating the importance of Web application testing and then outlining major techniques covered in subsequent sections. In Section 3 we cover graph and model based testing techniques, including finite state machine-based techniques. In Section 4, we briefly discuss mutation testing techniques. In Section 5, we present search based software engineering techniques, where testing problems are treated as optimisation problems. Section 6 is devoted to the discussion of some popular scanning and crawling techniques and present their application to security testing of Web applications. In Section 7, we present random testing, with the use of assertions as the primary oracle, and describe examples of how random testing can be applied to Web services. Fuzz testing is a form of random testing that generates invalid inputs with an aim of discovering defects that are severe and hard to detect. Section 8 is devoted to fuzz testing. We introduce some white-box fuzz testing techniques that make use of symbolic execution techniques introduced in Section 9. A black-box fuzz testing framework for JavaScript [6] will also be covered. Concolic testing, a technique that combines symbolic and concrete random execution to improve testing effectiveness, is covered in Section 9. We also show how they can be applied to testing dynamic PHP [7] and JavaScript applications using a number of examples. In Section 10, we discuss user session-based techniques and some of the ways to minimise the number of user sessions during testing. Lastly, in Section 11, we provide a summary of the different testing techniques and lay out future directions in which Web application testing research can proceed.

2. Motivation, challenges and overview of techniques

Many aspects of a Web application may be subject to testing, which has been a major challenge due to their heterogeneous nature. Web applications usually comprise different components that are typically implemented in different programming languages, application development frameworks and encoding standards. Additionally, as we stated above, compatibility testing has also become a major challenge with the increased availability of a number of popular browsers. Large Web-based software systems can be quite complicated and contain thousands to millions of lines of code, many interactions among objects, and involve significant interaction with users. In addition, changing user profiles and frequent small maintenance changes complicate automated testing. In the following 2 subsections we motivate the importance of testing with two challenges facing Web applications: *interoperability* and *security*. In Section 2.4, we then provide a quick overview of the major testing techniques which are also summarised in a number of tables for easy reference.

2.1. Interoperability

According to the World Wide Web Consortium (W3C),[1] the main international standards organisation for the World Wide Web (WWW), testing in Web applications is very significant. In order for the Web to reach its full potential, it is paramount that all the basic Web technologies are compatible with each other and allow any hardware and software used to access the Web to work together.[2] This goal is referred to as "Web interoperability" by the W3C. Two different implementations of a technology are compatible if they both conform to the same specifications. Conformance to specifications is a necessary but insufficient condition for interoperability; the specifications must also promote interoperability (by clearly defining behaviours and protocols). Therefore, in the case of Web applications and Web technologies, testing must be done to ensure that the overall functionality of a Web application conforms to the specification document(s) in addition to ensuring compatibility across different browsers (e.g., Chrome, Firefox, Internet Explorer and Safari) and platforms (e.g., different operating systems such as, Windows, Linux, Mac OS X, Android and iOS). Such an articulate version of testing will also help uncover contradictions, lack of clarity, ambiguity, and omissions in specification documents.

2.2. Security

Web applications are used by virtually all organisations in all sectors, including education, health care, consumer business, banking and manufacturing, among others. Thus, it is important to ensure that the Web applications developed are properly tested due to the importance and the sensitivity of the information stored in databases of such Web applications [8,9]. Thus, the security of Web application becomes an issue of critical importance. This is because Web applications can be

[1] http://www.w3.org/Consortium/. This site was last accessed on January 31, 2013.

[2] http://www.w3.org/QA/WG/2005/01/test-faq#why. This site was last accessed on January 31, 2013.

accessed by a large number of anonymous users and as a result, the information can be easily misused, possibly resulting in huge damages to the organisation and its clients.

Although it is important that Web applications are dependable, recent reports have indicated that in practice they are usually not. For example, one study of Web application integrity found that 29 of 40 leading e-commerce sites and 28 of 41 government sites exhibit some type of failure when exercised by a "first-time user" [10]. Similarly, another study by Kals et al. [8] showed that between 4% and 7% of randomly chosen Web site forms (from a list of 21,627 forms) were vulnerable to different categories of Cross-site scripting (XSS) and SQL injection attacks (more specifically, 6.63% to SQL injection, 4.30% to Simple XSS injection, 5.60% to Encoded XSS injection, 5.52% to Form-Redirecting XSS injection, see Section 6 for more details). Additionally, there have been recent cases in some high profile corporations, where lack of security in Web applications resulted in hackers gaining unauthorised access to the organisation's network and privileged information. For instance, the PlayStation Network of Sony Computer Entertainment was attacked in April 2011, resulting in hackers gaining access to the e-mail addresses, usernames, passwords, online user IDs and credit card details of nearly 70 million customers who were registered with Sony's PlayStation Network.[3] This large-scale breach of the security system of the PlayStation Network is believed to have cost Sony as much as $24 billion,[4] in addition to dealing a major blow to Sony's reputation worldwide.

2.3. Dynamics

Many non-trivial Web applications are divided into a server-side backend and a client-side frontend. The backend is responsible for data processing and persistence, and it often implements complex business logics. The frontend, traditionally concerned about the presentation of data, is becoming more and more sophisticated and rich in features. Dynamics are present in Web applications in several ways, and they bring unique challenges to the testing of Web applications.

Firstly, the Web applications themselves may be dynamic in nature. The so-called Web 2.0 [11] applications are characterised by their abilities to support not only static data presentation, but also interactive user participation and content creation. In these interactive Web applications, page contents can be updated by client-side scripts without a page refresh, made possible by languages and technologies such as JavaScript [6] and AJAX (Asynchronous JavaScript and XML) [12]. Such dynamic content generation mechanisms make capture-replay style of testing more difficult [13]. For example, the test driver needs to understand when page contents are ready to consult the test oracle for validating test results. The increasing prevalence of single-page Web applications[5] further amplifies the importance of dynamic Web application testing.

Secondly, predominant Web programming languages, including JavaScript [6], PHP [7], Python [14] and Ruby [15], are dynamic in nature. For example, JavaScript is the lingua franca for client-side scripting in Web applications. It is a powerful language with many advanced features, including dynamic typing and evaluation, functions as objects, and various forms of delegation. These features make it very challenging to thoroughly test a JavaScript application. For instance, a JavaScript application may accept many kinds of inputs, including responses from the server and user input from fields, which are structured as strings [16]. Therefore, a JavaScript testing tool must be able to discern the different kinds of inputs and handle them accordingly.

2.4. Overview of techniques

In this survey we broadly categorise Web application testing techniques into a number of groups, including those based on graphs and models, scanning and crawling techniques, search-based techniques, mutation testing, concolic testing, user session-based testing and random testing.

Each of these groups of testing techniques can be described along a number of dimensions, including a main purpose (to determine which technique should be used given the basic testing objectives), evaluation criteria, inputs and outputs, and criteria for stopping the test. For easy reference, these dimensions are highlighted in a number of tables in this subsection.

Table 1 summarises the main purpose of these testing techniques. Table 2 highlights the different testing techniques, evaluation criteria on the basis of cost-effectiveness, density of faults detected, and coverage. Table 3 describes the inputs, outputs and stopping conditions for each testing technique.

The *graph and model based testing* approach essentially creates a model of a Web application. Test cases are then derived on the basis of the model constructed. The test cases are generated according to either the all-statement (all statements must be covered/tested) or all-path (all paths/branches must be covered) coverage criterion. The *graph and model based approach* includes *finite state machine-based testing*, where a finite state machine depicting the model of the system is first constructed, from which test cases are then derived. A variant of *finite state machines* is the *probable finite state machines*, where transitions are associated with probabilities (this is similar to the probabilistic finite state machines discussed in [17, Section 7.1], where the shortest length message can be inferred from the data).

Mutation testing is aimed at detecting the most common errors that typically exist in a Web site or a Web application. In this form of testing, some lines of source code are randomly changed in a program to check whether the test case can detect the change. For example, the destination address on the client side, in an HTML form, may be replaced with an invalid address, or invalid files may be included in the server side of the program. If the test suite can detect the errors (i.e., testing has been properly conducted), then an error message will be displayed. This form of testing is mainly intended to ensure that testing has been done properly and also to cover additional faults which may exist in a Web site, and for which testing has not been performed.

[3] http://www.wired.com/gamelife/2011/04/playStation-network-hacked/. This site was last accessed on January 31, 2013.

[4] http://www.businessinsider.com/playStation-network-breach-could-cost-sony-24-billion-2011-4. This site was last accessed on January 31, 2013.

[5] http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php. This site was last accessed on November 15, 2013.

Table 1
The main purpose of each group of testing techniques and whether or not the techniques are automated.

Testing technique	Ref.	Automated	Main purpose
Model and Graph Based Testing	[19–21]	×	Create a model of the application to test
Mutation Testing	[22]	✓	Find out rare and most common errors by changing the lines in the source code
Search-based Testing	[23]	✓	To test as many branches as possible in an application via the use of heuristics to guide the search
Scanning and Crawling	[8,13,24,25]	✓	Detect faults in Web applications via injection of unsanitised inputs and invalid SQL injections in user forms, and browsing through a Web application systematically and automatically
	[26]	×	Detects navigation and page errors by systematically exploring pages and filling out form
Random Testing	[27–30]	✓	Detect errors using a combination of random input values and assertions
Fuzz Testing	[31–33]	✓	Test the application by passing in random, boundary or invalid inputs
Concolic Testing	[34,27,35,16]	✓	To test as many branches as possible by venturing down different branches through the combination of concrete and symbolic execution
User Session Based Testing	[36,9]	✓	Test the Web application by collecting a list of user sessions and replaying them
	[37,38]	✓	Reduce thetest suite size in User session-based testing

Table 2
The main evaluation methods of each of the testing techniques. The evaluation methods are indicated by a '✓' for each technique.

Testing technique	Ref.	Evaluation methods		
		Cost-effectiveness	Density of faults detected	Coverage
Model and Graph Based Testing	[19–21]	✓		✓
Mutation Testing	[22]		✓	✓
Search-Based Testing	[23]	✓	✓	✓
Scanning and Crawling	[13,8]		✓	
	[24]		✓	✓
	[25]	✓	✓	
	[26]			
Random Testing	[27,28]	✓	✓	✓
	[29]	✓		✓
	[30]	✓	✓	
Fuzz Testing	[31,32]	✓	✓	
	[33]	✓		✓
Concolic Testing	[34,16]	✓	✓	✓
	[27]		✓	✓
	[35]	✓	✓	
User Session-based Testing	[36–38]	✓	✓	✓
	[9]		✓	✓

Search-based software testing aims at testing a majority of the branches in a Web application. The main aim of this form of testing is to cover as many branches as possible and thus improve testing coverage. Usually, some heuristic is used to ensure that a large number of branches are tested and thus testing is sufficiently thorough.

Scanning and crawling techniques are mainly intended to check the security of Web applications. In such techniques, a Web application is injected with unsanitised input, which may result in malicious modifications of the database if not detected. The main idea is to detect any such vulnerabilities that a Web application may have. This is a very important form of testing, because, as discussed earlier, many Web site designers do not pay enough attention to security threats, thus making their Web site vulnerable to potential intrusion. This form of testing aims to improve the overall security of a Web site.

In *random testing*, random inputs are passed to a Web application, mainly to check whether the Web application functions as expected and can handle invalid inputs. A special case of random testing is *fuzz testing*, where boundary values are chosen as inputs to test that the Web site performs appropriately when rare input combinations are passed as input.

The major aim of *concolic testing* (concrete, symbolic testing) is also to cover as many branches as possible in a program. In this form of testing, random inputs are passed to a Web application to discover additional and alternative paths which are taken by the Web application as a result of different inputs. The additional paths are stored in a queue in the form of constraints, which are then symbolically solved by a constraint solver. The process continues until the desired branch coverage is achieved.

In *user session-based testing*, testing is done by keeping track of user sessions. In this case, a list of interactions performed by a user is collected in the form of URLs and name-value pairs of different attributes, which are then used for testing. Due to the large number of user sessions that can result when a user interacts with a Web site, there are several techniques for reducing the number of sessions

Table 3
The main inputs, outputs, and stopping conditions for each testing technique.

Testing technique	Ref.	Inputs	Outputs	Condition to stop testing
Model and Graph Based Testing	[19,21]	Model of the application	Regular expressions from which test cases can be created	Depends (e.g., all path/all statement criteria)
	[20]	Lower level finite-state machine (FSM) of the application	An application-level FSM from which test cases can be generated	Depends (e.g., all path/all statement criteria)
Mutation Testing	[22]	A program and mutation operators	Mutated program created as a result of applying the mutation operators	Required mutation operators are applied
Search-based Testing	[23]	Mutation of application inputs as generated by different heuristics such as hill climbing, simulated annealing and evolutionary algorithms	A test suite with the aim of maximise branch coverage	A pre-determined, fixed number of test executions
Scanning and Crawling	[24,13,8,25]	Unsanitised user inputs to crash the Web application (e.g., the database or force the user to enter unsafe Web sites)	Type of defects and number of defects	Until all the forms for a given Web application is injected with different forms of unsanitised user input
	[26]	A starting URL	Type of defects and number of defects	A maximum depth of exploration is reached
Random Testing	[27]	WS-CDL Web service choreography specifications	Test oracles (a test suite including assertions)	The `stop` instrument in the control flow graph is reached
	[28]	State machine models of WSDL Web service descriptions	A test suite, and its execution and visualisation	Depends (e.g., all path/all statement criteria)
	[29]	A JavaScript program	A set of test cases and their execution results	A pre-determined, fixed number of test cases
	[30]	A JavaScript program	Test oracles in the form of contracts and random test cases based on contracts	All contracts exhausted
Fuzz Testing	[31]	Random user inputs which test rare or unlikely system behaviour (e.g., values near the boundaries)	Test cases with oracles	Depends (e.g., all path/all statement criteria)
	[32]	A JavaScript program and a benign input	A test suite and the identified potential client-side validation vulnerabilities	All identified data flows exhausted
	[33]	A Program, a grammar and an initial input	A test suite and the identified defects	All generated inputs exhausted
Concolic Testing	[34,27,35,16]	Concrete inputs from symbolically solved previous iteration, start with random initial input	Path constraints at each iteration	All constraints exhausted
User Session-based Testing	[36,38,9]	User sessions	A combination of URL and the parameters to be passed to the server	Depends (e.g., certain selected user sessions have been tested or reasonable coverage has been achieved)
	[37]	User sessions	An updated concept lattice and a updated test suite	Continue till all the user sessions to be tested have been exhausted and the test suite and the lattice cannot be modified anymore

to test, such as lattice construction, batch test suite reduction, incremental reduced test suite update, and test case reduction through examining URL traces. One of the aspects of Web application testing that we do not cover in this survey is *usability testing* [18]. Usability testing is primarily a black-box testing technique. The major aim is to test how users use an application and discover errors and/or areas of improvement (intended to make the product more intuitive and user-friendly). Usability testing generally involves measuring how well users respond in four main areas while using the application: efficiency, accuracy, recall, and emotional response. The results obtained from the first test are usually treated as a baseline against which all subsequent tests are then compared to indicate improvement. Generally speaking, in the case of Web applications, such usability testing would involve the testing of, e.g., (1) the ease of using the application, (2) the layout and appearance of the Web application on different devices such as desktops, laptops, and mobile systems, and (3) whether different messages displayed during the application are sufficient and appropriate.

3. Graph- and model-based white-box testing techniques

These testing techniques start with constructing a graph or a state machine model of the Web application and then generate test cases from these models.

3.1. Graph-based testing

The most popular white-box graph based testing approach is the one proposed by Ricca and Tonella [19], which creates a graph-like model in which nodes in the graph represent Web objects such as Web pages, forms and frames, and edges represent the relationship between these objects (e.g., submit, include, split and link).

To generate test requirements and test cases, a regular expression to match the graph is generated. For example, Fig. 1 depicts the graph for a sample Web application. The regular expression "$e1e2+e1e3^{*}+e4e5^{*}$", where a "$*$" indicates that a transition may be visited any number of times and a "$+$" indicates choice, can then be used to generate test requirements by identifying the set of linearly independent paths that comprise it, and applying heuristics to minimise the number of requirements generated. A linearly independent path is defined as the path in which at least one edge has not been traversed earlier while constructing the paths and all the linearly independent paths together test all the edges of the graph at least once.

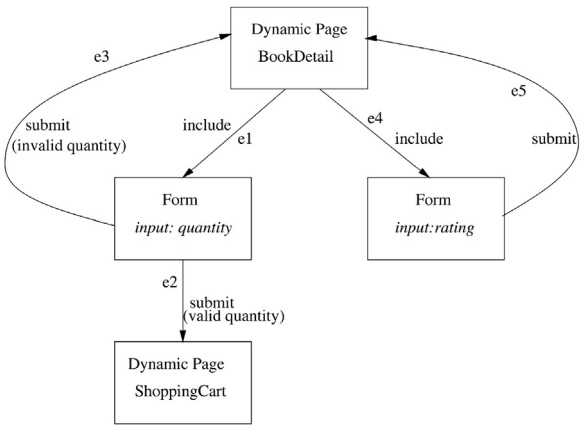

Fig. 1. The graph model of a sample Online Book Shopping Web application from [19, Fig. 2].

The test cases are sequences of Web pages to be visited together with their corresponding values (generated from the path expressions). There are two versions [19] of this implementation:

- Complete test requirements with ad hoc selection of inputs: This white-box technique attempts to match the methodology presented in [19]. Test cases are generated from the path regular expressions and the following assumptions are made: (1) only linearly independent paths are tested, (2) we exercise forms that are included in multiple Web pages, but perform the same functionality independent of context (e.g, provide search capability) from only one source, and (3) ignore circular paths which link back to the starting page (included just to facilitate navigation within a page). Then the forms are filled and test cases are generated.
- Complete test requirements with formalised selection of inputs: This technique uses boundary values as inputs, and utilises a strategy for combining inputs inspired by the "each condition/all conditions" strategy [39]. The test suite contains test cases such that the test cases for each form comprise tests of empty values for all variables and one additional test case in which all the variables have values assigned. For the test cases that consider just one variable, the values are selected based on the boundary conditions for such variables. For the test case that includes all variables at once, one random combination of values is selected. The objective behind this strategy is to add formalism to the process of inputting data into the forms, as recommended in one of the examples in [19].

One of the main limitations of this approach by Ricca and Tonella [19] is that the construction of the graph has to be done manually. Therefore it is very difficult to automate the process. In addition, only linearly independent paths are tested. The advantage of this approach seems to be that there is not much overhead involved in obtaining the regular expressions after a model of the system is constructed (manually). Moreover, the testing involves inputs chosen randomly, thus making the process simple to implement.

3.2. Finite state machine testing

A finite state machine usually has a finite set of states S, a finite set of inputs I, a finite set of outputs O, a transition function T which determines the transition from the current state s_1 to a next state s_2, depending on the input i_1, and an output function O, which determines the output produced by a transition.

A few methods have been proposed for deriving test cases for software systems from finite state machines (FSMs) [40]. Finite state machine-based approaches have also been applied to Web testing because a Web application is essentially a system in which transitions occur from one Web page to another and outputs are produced according to the inputs/actions at and the current state.

Test cases for testing a Web application can be generated from FSMs. The methodology required to generate test cases from FSMs is described in [20]. A simple Web application can face the *state space explosion problem* as there can be a very large number of possible inputs to a text field, a large number of options (e.g., checkboxes, links) can be available on a particular page, and a large number of different orders in which the options can be selected. For example, for a simple set of 5 questions each containing 4 possible options (in the form of checkboxes), there can be 4^5 (1024) different combinations of user selected options. Thus, the FSMs must be expressive enough to test the system effectively and also be small enough to be practical [20].

Hierarchical FSMs [20] have been employed to alleviate the state space explosion problem, by reducing the number of states and transitions in an FSM. The bottom level FSMs are formed from Web pages and parts of Web application known as *logical* Web pages, and the top level FSM represents the entire Web application. A *logical* Web page may either be a simple physical Web page or an HTML form which accepts inputs from the user and sends it to a different software module. The logical Web pages (LWP) can be easily extracted because these are embedded with HTML "Form" tags.

Next, in order to generate test cases manually from the Web applications, the following four steps are performed [20]:

1. Partitioning the Web application into *clusters*, where a *cluster* comprises software modules and Web pages which implement a certain logical function. Clustering is done to identify the different layers of abstraction. At the top level, clusters are abstractions that implement functions that can be identified by users. At lower levels, clusters are a set of Web pages and software modules which communicate with each other to implement some user-level functions. At the lowest level, clusters may be individual Web pages and software modules that represent single major functions themselves. Individual clusters can be identified from the layout of the site navigation, coupling relationships among the components, and the information that can be obtained from site design [20]. This process is manual, and as a result, the clusters and the entire partitioning process can have an impact on the resulting tests [20].
2. Extracting Logical Web Pages (LWP). This process can be automated, because HTML Forms that accept input data from the user and send it to a back-end software module are embedded with HTML "Form" tags. The identification of LWPs can, therefore, be carried out by extracting these HTML tags.
3. Building FSMs for each cluster. A bottom-up approach is followed to construct these FSMs. First, the FSMs are generated from bottom-level clusters that contain software modules and Web pages (i.e., no clusters). Next, higher-level cluster FSMs are built by aggregating lower-level FSMs. Each state or node in these higher-level FSMs represents a lower-level FSM. This process is completed manually.
4. Building an application FSM for the entire Web application. Lastly, an application finite state machine (AFSM) defines a finite state model of the entire Web application, in which the edges represent the navigation or links between different Web pages in different clusters. Each FSM is assumed to have a single entry and exit node ("dummy" or extra nodes may be added to guarantee this requirement). The final result of this partitioning is a collection of autonomous (separate but interacting) finite state machines with the following two properties. First, they are small enough to efficiently allow test sequences to be generated. Second, they clearly define the information that propagates among the FSMs. This process is accomplished manually.

The state space explosion problem in finite state machines can be dealt with more effectively by defining a BNF grammar for the input constraints on the Web application. For input values, [20] defines five broad choices, namely, R (if the input is *required*, i.e., the user has to enter a value to transition from one state to another), $R(parm = value)$ (if the input is required and can only be chosen from a subset to transition from one state to another), O (if input is optional), C_1 (if the user only needs to *select* one input from a given list of inputs, e.g., a radio button) and C_n (if the user needs to select multiple values from a list, e.g., check-boxes). In order to define the *order* of values, [20] defines two symbols, S (if the values must be entered in a particular order) and A (if the values can be entered in any order). The *types* of inputs can be further broken down into "text" and "non-text". FSMs with input constraints are called "annotated FSMs". The advantage of "annotated FSMs" over normal FSMs is shown clearly in Fig. 2(a) and (b). Assuming that there is a Web application with input fields "username" and "password", where the username and password must differ from each other and there are only 3 possible choices for usernames and passwords, namely "a", "b" and "c", then the normal FSM with different states and transitions will be constructed as shown in Fig. 2(a). But, if annotated FSMs are used, then the different states and transitions can simply be replaced with the "annotated constraints" (such as "R", "S" and "A" to test for the order of entered values as well as for the actual values entered).

The FSMs defined above are then used to generate tests. Tests are generated as sequences of transitions and the values for inputs are then selected randomly. A test sequence is a sequence of transitions in an FSM and the associated constraints.

The main limitation of this approach is that the finite state machines have to be constructed manually. Thus, the process is difficult to be fully automated. The advantage of this approach is in the reduction in the number of states in an FSM, which helps solve the state space explosion problem as discussed above. Also, the extraction of the logical Web pages (LWPs) can be done automatically.

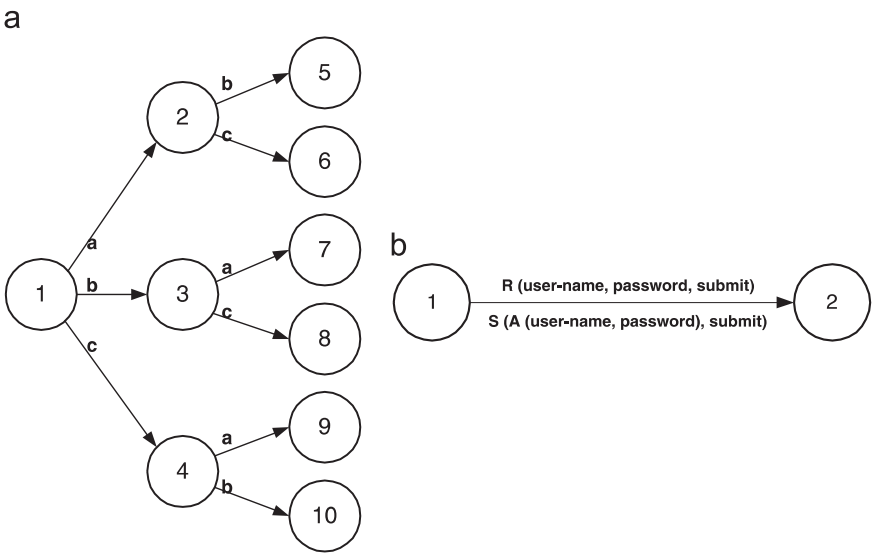

Fig. 2. A demonstration of the advantages of an annotated FSM over a normal FSM, as given by [20, Fig. 8]. (a) Normal FSM for a simple Web application with fields username and password. (b) Annotated FSM for the same Web application.

The use of annotated FSMs also makes it easier to derive a model of the complete Web application incrementally.

3.3. Probable FSM

An extension of the FSM described above is the *probable* (probabilistic) FSM. A method for generating test cases from the probable FSM is described in [21], where the probabilistic FSM is constructed manually, and the test sequences are selected on the basis of probabilities of a given sequence. In such an FSM, in addition to the states and transitions, a probability is also assigned to each transition. In this case, the transition function T is redefined to include the current state, the transition t_1 and the probability associated with t_1. Thus, the transition function $\delta(s_1, t_1, p_r) = s_2$ means that if the current state is s_1 and there is a transition t_1 from s_1 with the probability p_r (such that $0 \le p_r \le 1$), then there will be a transition from s_1 to s_2. The sum of all the values from a given state may not add up to 1. This is because the individual probabilities of different transitions represent the probability with which it is possible to get to another state (e.g., user session). Thus, there are other events, of very low or negligible probability, which are not usually modelled in the probable finite state machine. The proposed PFSM usage model can capture information about control flow, data flow, transaction processing and associated probabilistic usage, and criticality information. An example probable FSM [21] with transitions and states is shown in Fig. 3.

A test case is defined as any path from the source node (start state) to the target node (final state). In order to generate test cases, a list of all the path transitions from the source node to the target node is generated. Then probabilities are calculated for all the individual paths that have been generated (by multiplying the individual probabilities for the transitions). A specific threshold value is also obtained. The test suite then comprises all the paths which have a probability higher than the threshold value.

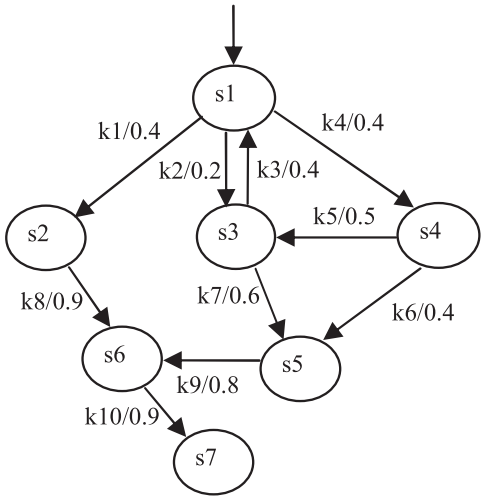

Fig. 3. An example probable FSM as described in [21, Fig. 1].

For example, in the probabilistic finite state machine depicted in Fig. 3, there are at least four potential paths from state s_1 to s_5, namely:

Path A: $k_4 \rightarrow k_5 \rightarrow k_7$ $(s_1 \rightarrow s_4 \rightarrow s_3 \rightarrow s_5)$
Path B: $k_4 \rightarrow k_6$ $(s_1 \rightarrow s_4 \rightarrow s_5)$
Path C: $k_2 \rightarrow k_7$ $(s_1 \rightarrow s_3 \rightarrow s_5)$
Path D: $k_4 \rightarrow k_5 \rightarrow k_3 \rightarrow k_2 \rightarrow k_7$ $(s_1 \rightarrow s_3 \rightarrow s_1 \rightarrow s_3 \rightarrow s_5)$

The probabilities for each of these individual paths are simply the multiplications of the individual probabilities of the different transitions. Thus, as per Fig. 3, the probabilities are as follows:

Path A: $0.4 * 0.5 * 0.6 = 0.12$
Path B: $0.4 * 0.4 = 0.16$
Path C: $0.2 * 0.6 = 0.12$
Path D: $0.4 * 0.5 * 0.4 * 0.2 * 0.6 = 0.0096$

Now, only the test cases which meet the threshold value are selected. Thus, assuming that the threshold value was set to 0.1, only Paths A and B will be selected as the test cases for the test suite. A high threshold value can be selected to test only the most frequently used operations and gradually lowered to involve the rarer and more unique operations and ensure the satisfactory coverage or reliability for a wider variety of operations. Thus, the threshold value can be modified to control the number of test cases generated and its coverage.

The limitation of this work is that the probable FSM cannot be generated automatically and thus the testing process will be slow since the FSM construction has to be done manually. This work also does not explain thoroughly as to how the probabilities are assigned to different transitions from different states. Also, the FSMs represent an incomplete model of the system, since only the most "likely" transitions are modelled, and the unlikely events are not included in the FSM. Thus, it would be very hard to test for "rare events" which may be crucial to the security of Web applications.

4. Mutation testing

Mutation testing is a form of testing in which a program *P* is taken as input. A modified version of the program *P* is then created by applying certain *mutation operators* to the original program. Test cases are designed with the aim of detecting program modifications. The modified versions of the program are called *mutants* [41–43] and if a test case can detect a mutant (i.e., the line of code where the mutation operator has been applied), then the test case is said to *kill the mutant*. Otherwise the mutant stays *live*.

Praphamontripong and Offutt [22] implement mutation testing for Java Server Pages (JSP) and Java Servlets. The mutation operators are defined specifically for these applications and are implemented in a tool called muJava [44]. muJava automatically creates mutants and allows tests to be run against the mutants. Tests are created manually as sequences of requests [22].

11 new mutation operators are defined specifically for Web applications [22]. These mutation operators are grouped into two categories, (1) operators modifying HTML applications and (2) operators modifying JSP applications. Some of the operators for HTML include, e.g., "simple link replacement (WLR)" which replaces the "`<a href=>`" attribute value with another address in the same domain. The "W" in WLR indicates that the mutation operator deals with Web-specific features, "R" indicates that the operator *replaces* some field and "L" indicates that it exercises the *links* in a Web application (e.g., non-existent or incorrect URLs). Similarly, mutation operator exists for (a) *deleting* a link ("simple link deletion (WLD)", where "D" indicates that the operator deletes a field), (b) replacing the destination address of an HTML *form* ("form link replacement (WFR)"), (c) replacing "`GET`" requests with "`POST`" requests and vice versa ("transfer mode replacement (WTR)"), and (d) replacing and deleting the form values of type *hidden*. Also, in order to detect server side faults, Praphamontripong and Offutt [22] create mutation operators for replacing and deleting server side "`include`" statements (which basically describes what other files need to be included in a particular application). Mutation operators for JSP include operators which change the forwarding destination of a redirect transition specified in "`<JSP:forward>`" ("redirect transition replacement (WRR)"), and deleting the destination address ("redirect transition deletion (WRD)").

Praphamontripong and Offutt [22] apply testing on a moderate-sized application called the "Small Text Information System (STIS)" [45]. STIS comprises 18 Java Server Pages (JSPs) and 5 Java bean classes and stores the information in a MySQL database. Praphamontripong and Offutt [22] applied mutation operators to only the JSPs and excluded 2 JSPs. A total of 219 mutants of the application were tested, where the mutants were created by using the mutation operators described above. The total number of *live mutants* in the application (i.e., the mutants which reveal different faults) and which were undetected by the tool muJava are 29 [22]. Most of these undetected faults were related with replacing the form values of type *hidden* (23 out of 29). Thus Praphamontripong and Offutt [22] show that mutation testing could be used to reveal a large number of faults in a Web application (nearly 86% in this case). A total of 147 hand-seeded faults are also planted in the application, out of which 118 were detected ($\sim$80%) [22].

One of the main advantages of mutation testing approaches for Web applications is that it tests for most crucial errors which are likely to occur in a Web application. For example, a lot of the server errors occur in a Web application either due to some invalid form attributes, missing files or as a result of not validating user inputs properly. The mutation testing technique proposed by Praphamontripong and Offutt [22] is particularly effective in detecting such defects which may occur on the server side. Similarly, on the client-side of the application, a large number of errors are due to broken links (i.e., errors in the destination links), missing files or invalid HTML. Again the mutation testing approach [22] can be effectively applied for these defects.

5. Search based software engineering (SBSE) testing

Search Based Software Engineering (SBSE) is an approach that treats software engineering problems as optimisation problems whose solutions require searching through a state space [23]. The possible solutions need to be encoded in a way that makes similar solutions proximate in the search space. A fitness function is then defined which is then used to compare possible solutions. Hill climbing is an iterative incremental algorithm often used in SBSE and found to be effective for testing [46]. In hill climbing, a random solution is first chosen and evaluated and then the nearest neighbours (determined by some heuristic, e.g., distance) are evaluated. This process is iterated by changing a single element in the solution and thus obtaining new solutions. If the value of the fitness function of the new solution is better than the value of fitness function of the older solution, then the newer solution replaces the previous one. This process is repeated iteratively until no further improvements

can be made to the solution. Thus, the algorithm aims to find a solution which maximises the fitness function (or the heuristic). This approach is simple and fast. However, it is dependent on the randomly chosen starting point of the solution.

For the hill climbing algorithm, Korel [47] introduced the Alternating Variable Method (AVM) into the search process. This method changes one variable while ensuring other variables remain fixed. Branch distance is used to measure how close an input comes to covering the traversal of a desired branch. When the execution of a test case does not converge on the target branch, the *branch distance* expresses how close an input came to selecting the correct path/branch at a particular level and satisfying the predicate [48]. It also helps in determining the level along the target branch of the predicate at which control for the test case went "wrong"; i.e., how close the input was to descending to the next branch level. The branch distance is computed using the formula $|offset-1|+K$, where K is the constant added when an undesired, alternate branch is taken by the test case. The lower the absolute value of $offset-1$, i.e., the closer the value of *offset* is to 1, the closer the test case is to traversing along the correct target branch [48]. A different branch distance formula is applied depending on the type of relational predicate. In the case of relational equals, the branch distance is equal to $|a-b|+K$. The formulas for different branch distances depending on the type of relational predicates are described in [49].

If the changes to a variable affect branch distance, a larger change is applied in the same direction in the next iteration. In case, as a result of applying this change, a false local optimum is chosen, the search is re-started at the previous best solution seen. The process continues until the branch is covered or no further improvement is possible.

Alshahwan and Harman [23] apply Search Based Software Engineering to testing PHP Web applications. The main aim of this technique is to maximise the branch coverage of the application. 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 Korel's Alternating Variable Method (AVM) in addition to constant seeding.

Several issues affect the application of search based techniques to Web applications, such as *Dynamic typing* (e.g., in different languages such as Ruby and PHP, variables are dynamically typed, which makes it hard to determine the type of variables used in predicates, which may in turn lead to problems when deciding which fitness function to use). Another important issue is *Interface Determination*, which means that there is no way of determining the interface in different PHP or JSP applications. In other words, there is no way of knowing how many inputs are required for the application to execute. Other problems also include client-side simulation of dynamic Web pages and dynamic *server-side* `include` statements.

The algorithms for test data generation in [23] are based on hill climbing using Korel's AVM [47]. When a target branch is selected, AVM is used to mutate each input in turn while all other inputs remain fixed. When the selected mutation is found to improve fitness value, the change in the same direction is accelerated. To avoid overestimating (or over shoot), the change is decelerated when the fitness function nears 0. Branches which have been reached but not covered are then targeted in subsequent iterations. That is, a branch is reached if its immediately controlling predicate is executed, while a branch is covered if the branch itself is traversed. The algorithm(s) attempt to cover a branch only when it is reached, i.e., all transitively controlling predicates on some path have been satisfied. This technique is called an *explorative approach* [23]. At each iteration the algorithm also keeps track of *near miss* input values. A near miss input vector results in fitness improvement for a branch other than the targeted branch. Near misses are used in place of random values when initialising a search to cover that branch. This approach is called Near Miss Seeding (NMS).

The fitness function employed in this approach is similar to that used by Tracey et al. [49]. This means that for a predicate "$a\ op\ b$" where *op* is a relational operator, fitness is zero when the condition is true and the absolute value of $|a-b|$ when the condition is false. A fitness function has the value 0 if the test case covers the desired branch of the program. The main aim of the technique is to minimise the fitness function values throughout the search process. The fitness function value is incremented in a similar technique as Tracey et al. [49]. That is, if the test case is incorrect, then the value of the fitness function k is incremented by 1. For strings, Levenshtein distance [50] is used as a fitness function, following Alshraideh and Bottaci [51]. The Levenshtein distance is the minimum number of insert, delete and substitute operations needed to convert one string to another string [23].

Each execution of a test case returns a list (F) of all branches in that execution and the branch distances. For every branch (B) that recorded an improvement in the branch distance, the list, F, is used to update a coverage table (C), and the resulting test suite (T). A list of branches, known as the *work list*, that have been reached/traversed is extracted from the coverage table, and is then processed in an attempt to cover it. To start, the database is initialised and the user logs into the application. The input vector is then constructed using the analysis data. Values for variables are initialised to values that caused a particular branch to be reached, and random values are selected for any additional input variables. The input variables are mutated one at a time, and the process continues until the branch has been traversed or no further improvements to the fitness function are possible. The *mutation* algorithm is quite simple and is described in [23]. Initially, if no input was selected for mutation, or the last mutation had no effect on the branch distance, a new input variable is selected. If the branch distance is increased as a result of performing a mutation, then a new mutation operator is selected. Conversely, if the branch distance decreased as a result of the mutation operation, the operation is accelerated [23].

Alshahwan and Harman [23] developed a tool called the "Search based Web Application Tester" (SWAT) to implement this approach and embed it within an end-to-end testing infrastructure. SWAT's architecture is illustrated in Fig. 4. The tool is composed of a pre-processing component, the Search Based Tester and the Test Harness [23]. The description of the architecture is given below.

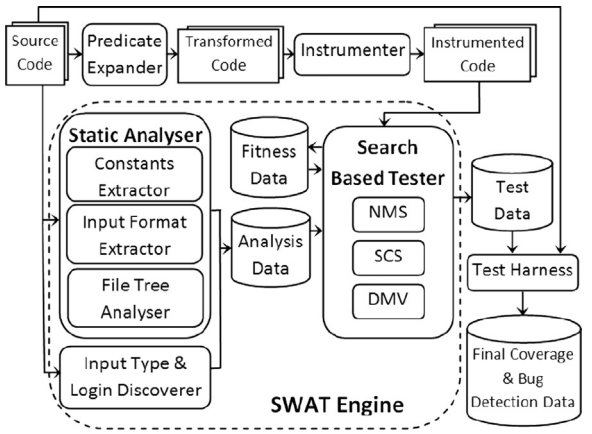

Fig. 4. The architecture of the SWAT system as described in [23, Fig. 2].

The original source code is transformed through the *Predicate Expander* and *Instrumenter*. In the resulting transformed version of the code, predicates with logical operators are expanded and control statements are modified to calculate fitness and the original behaviour of the predicates. The code is also instrumented to collect run-time values to be used in subsequent *dynamically mined value* seeding. The *Static Analyser* performs the analysis needed to resolve issues that are associated with the standard Web scripting and development languages, such as Ruby, JavaScript, JSP, ASP, PHP, etc. Some issues may include the distinction between different data types (such as Integer and String) in dynamically typed languages, the absence of an explicit header that specifies how many inputs a program expects and/or their types, handling dynamic `include` statements in PHP applications, and identifying top-level pages when trying to generate the test data.

The results obtained from the above step are stored in the *Analysis Data* repository and used later by the *Search Based Tester*. The *Constant Extractor* mines the code for constants to be used in subsequent *Static Constant Seeding*. The *Input Format Extractor* analyses the code to extract the input vector comprising the values for input variables. The *File Tree Analyser* then generates a tree. In this tree, the nodes are used to represent files and edges represent include relationships. This is then used to determine the top-level test units to be processed. The *Input Type and Login Discoverer* performs a conjunction of static and dynamic analysis to infer the types of input variables and identify the login process. The results of this the *Input Type and Login Discoverer* need to be augmented manually because the technique for type inference cannot infer types for all input variables. The *Login Discoverer* is used to dynamically extract the variables that need to be set (i.e., have a value) during *login*, such as the variables used to store the username, password, and the login URL. The concrete values for "username" and "password" are provided to the tool. The *Test Harness* uses the test data generated to execute the tests on the original source code in addition to producing coverage and bug data. When a test case is executed, the generated HTML and the Web server's error logs are parsed for PHP execution errors. This different components of this tool are implemented using a combination of different languages, namely, Perl, Java, and Stratego/xt (a program transformation language and PHP-Front provides libraries for Stratego/xt supporting PHP).

The main advantage of Search Based Software Engineering is that testing is complete and done thoroughly, with the major aim of improving branch coverage. This is evident since the algorithm aims to cover branches which were not covered in a previous iteration. The limitation would be that the algorithm would probably be slow as compared to other simpler testing techniques such as mutation testing, model based testing, or random testing.

6. Scanning and crawling techniques

Security vulnerabilities represent serious risks for Web applications. In a lot of applications, they result from generic input validation issues. Examples of such vulnerabilities are SQL injections and Cross-Site Scripting (XSS) attacks. The majority of these vulnerabilities are easy to detect and avoid if the programmers are aware of the potential pitfalls. However, many Web developers do not focus on security issues, which leaves the Web applications vulnerable to malicious attacks. As a result, many Web sites on the Internet suffer from insufficient security tests and checks.

Scanners are tools which detect these errors by injecting invalid inputs into a Web application and then determining what type of errors exist according to the behaviour of the Web application. Crawlers are tools that browse the Web and collects information in a predefined and automated manner. There are many different scanning and crawling techniques that are used for detecting vulnerabilities in Web applications. In most cases, the vulnerability of a Web application is detected by injecting faults into the Web application. This is a reliable form of testing Web sites for security vulnerabilities and can thus be used to detect the type of bugs present as well as the number of bugs present in a Web application.

The main advantage of scanners is that it helps in detecting the bugs which the programmer usually does not think of testing when designing Web applications. As was discussed earlier, a large number of Web sites do not perform proper Web site form validation and this may result in unwanted reads from and writes to the database. Thus, this may result in the breach of sensitive private information on a large scale, especially for popular applications such as banking and e-commerce Web sites. Additionally, such bugs may also result in the loss of reputation of a company. Thus, detecting such bugs helps in improving the quality and security of the Web site and helps in preventing major economic losses.

The basic idea behind scanning is that some *unsanitised* input is injected into HTML forms, which is then sent to the server. If the Web application has proper validations and performs proper input sanitation for user data, then it will behave normally. Otherwise, there may be breaches in security and severe implications such as writing unsafe values to the database and breaching of privatised data. Scanners can be grouped into two broad categories, black-box and white-box scanners. Out of these two, black-box scanners are more popular due to the limitations of the

white box scanners, in particular due to the heterogeneous programming environments (i.e., many programming languages used to develop different parts of the Web application). Additional factors which limit the effectiveness of white-box scanners include the complexity of applications, which incorporate databases, business logic, and user interface components. In practice, black-box vulnerability scanners are used to discover security problems in Web applications. These tools operate by launching attacks against an application and observing its response to these attacks.

We start with an introduction to two prominent forms of attacks, cross-site scripting (XSS) and SQL injection in Section 6.1. In Section 6.2 we present a number of black-box scanning techniques and systems. Dynamic AJAX-based Web applications present unique challenges to testing. We discuss these challenges and survey some JavaScript and AJAX crawlers and show how they can be used to test AJAX-based Web applications in Section 6.3.

6.1. XSS and SQL injection detection techniques

SQL Injection (SQLI) and cross-site scripting (XSS) attacks are forms of attack in which the attacker modifies the input to the application to either read user data or trick the user into executing malicious code which may corrupt a large collection of different user records (e.g., a database). The serious attacks (also called second-order, or persistent, XSS) enable an attacker to write corrupt data into the database so as to cause subsequent users to execute malicious code. Common approaches to identifying SQLI and XSS vulnerabilities and preventing exploits include defensive coding, static analysis, dynamic monitoring, and test generation. Each of these techniques has their advantages and drawbacks.

Web applications usually read in user data which is then sent to the server side for processing. This data can then be used as parameters to SQL queries on the server side. Therefore, it is important to sanitise the user input because if the data is not properly processed prior to SQL query construction, malicious patterns that result in the execution of arbitrary SQL or even system commands being injected. Assume there are two field variables named "`userName`" and "`passWord`" in a "`login`" form. Then, in order to check for a valid user login, an SQL query can be constructed (after these two fields are sent to the server side) as follows [24, pp. 150]:

```
$userName = $_GET ['$userName']; //PHP GET request
$passWord = $_GET ['$passWord'];
//SQL Query
SQLQuery = "SELECT * FROM Users WHERE (UserName='" +
    $userName + "') AND (Password='" + $passWord + "');
    "
if (GetQueryResult(SQLQuery) = 0)
  validLoginUser = false;
else
  validLoginUser = true;
```

This code checks if there are any rows returned from the database with the user entered "`userName`" and "`passWord`" fields. If 0 rows are returned, then this means the login is invalid, otherwise it is valid. However, if the user input is not sanitised, a malicious hacker will be able to enter values for both fields such as `X' OR '1'='1'`. Now, this results in the SQL statement being converted to:

```
SQLQuery = "SELECT * FROM Users WHERE (UserName='X' OR
    '1' = '1') AND (Password='X' OR '1' = '1')";
```

Since the condition `'1'='1'` will always evaluate to `true`, the entire condition in the `WHERE` clause evaluates to `true` and no checking is done for the user entered values. As a result of executing this query, information about all users will be returned, which is a serious security breach.

As with SQL injection, cross-site scripting is also associated with undesired data flow. In order to understand it, a following scenario can be provided. A Web site for selling computer-related merchandise holds a public online forum for discussing the newest computer products. Messages posted by users are submitted to a CGI (Common Gateway Interface) program that inserts them into the Web application's database. When a user sends a request to view posted messages, the CGI program retrieves the messages from the database, generates a response page, which is then sent to the Web browser used by the client of the Web application. In this scenario, a hacker can post messages containing malicious scripts into the forum database. When other users view the posts, the malicious scripts are delivered via the response page and can be spread on a user's machine as a result of them using the Web application [52].

Most browsers enforce a Same Origin Policy[6] that limits scripts to accessing only those cookies that belong to the server from which the scripts are delivered. In this scenario, even though the executed script was written by a malicious hacker, it was delivered to the browser on behalf of the Web application. Such scripts can therefore be used to read the Web application's cookies and break its security mechanisms.

A tool for detecting such SQL injections and cross-site scripting, Web Application Vulnerability and Error Scanner (WAVES), is proposed [24]. In WAVES, the crawler (for exploring a Web site) tries to search for the existence of links inside a Web page by detecting HTML anchor tags, framesets, meta refresh directions, client-side image maps, form submissions, JavaScript event generated executions, JavaScript variable anchors and checking for JavaScript redirections and new windows. In addition, the crawlers act as interfaces between Web applications and software testing mechanism and allow the application of testing techniques to Web applications. WAVES performs an event-generation process to stimulate the behaviour of active contents. This allows WAVES to detect malicious components and assist in the URL discovery process. During stimulation, JavaScripts located within the assigned event handlers of dynamic components are executed, possibly revealing new links.

The WAVES architecture is represented diagrammatically in Fig. 5. The main purpose of the *Injection Knowledge Manager* (IKM) is to bypass the existing validation procedures in the Web application by producing variable candidates. This knowledge can also be used during the crawl process. When a crawler crawls through a form, it sends a query to the IKM. The data produced by the IKM is then submitted by the crawler to the Web application for discovery of further back-end pages, i.e., deep page discovery. In order to make the crawl process faster, a URL hash is implemented. This completely eliminates disk access during the crawl process. The global bottlenecks at the URL hash are further reduced by the presence of a distinct 100-record cache. This implementation strategy is similar to the one described in [53].

In WAVES, a sample site was established to test several academic and commercial crawlers, including Teleport,[7] WebSphinx [54], Harvest [55], Larbin,[8] Web-Glimpse [56], and Google. None were able to crawl beyond the fourth level of revelation (which is nearly about one-half of the revelation capability of the WAVES crawler, which is 7). Revelations 5 and 6 are the result of WAVES ability to interpret JavaScripts. Revelation 7 refers to link-revealing JavaScripts, but only after different user-generated events such as “onClick” and “onMouseOver”.

6.2. Black-box Web vulnerability scanners

Black-box Web application vulnerability scanners are automated tools that test Web applications for security

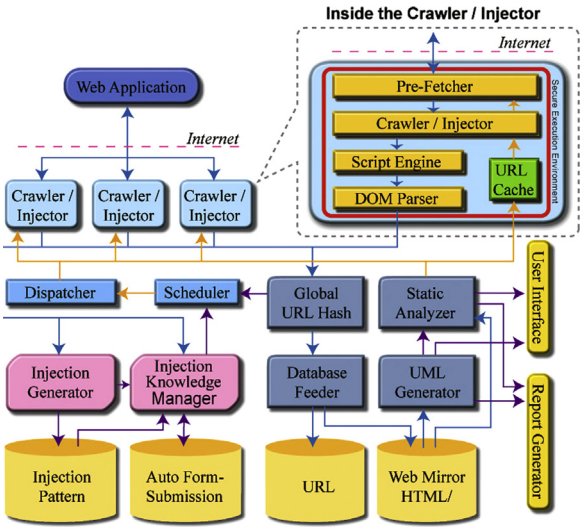

Fig. 5. A diagram showing WAVES architecture as described in [24, Fig. 7].

vulnerabilities. Black-box scanner does not have access to source code used to build the application. While there are intrinsic limitations of black-box tools, in comparison with code walk through, automated source code analysis tools, automated black-box vulnerability scanners also have advantages. Black-box scanners mimic external attacks from hackers, provide cost-effective methods for detecting a range of important vulnerabilities, and may configure and test defenses such as Web application firewalls. The effectiveness of a black box scanner depends on three factors, namely:

1. whether the scanner can detect key vulnerabilities of interest to the Web developers, i.e., the class/type of vulnerabilities detected by scanners,
2. effectiveness of the scanner in detecting faults, and
3. whether the vulnerabilities detected by scanners are representative of the general vulnerabilities of Web applications.

Bau et al. [25] performed a comparative study on 8 well-known commercial scanners and tested them on well-known Web applications such as phpBB,[9] Drupal[10] and Wordpress.[11] It was discovered that all of them had vulnerabilities. A custom application was described [25] to measure elapsed scanning time and scanner-generated network traffic.

Bau et al. tested the scanners for false positive performance and vulnerability detection. They found that the vulnerabilities tested most extensively by scanners are, in ascending order, Information Disclosure, Cross Site Scripting (XSS), SQL injection, and other forms of Cross Channel Scripting (XCS). This vulnerability distribution for faults is

[6] http://www.w3.org/Security/wiki/Same_Origin_Policy. This site was last accessed on January 31, 2013.

[7] http://www.tenmax.com/teleport/home.htm. This site was last accessed on January 31, 2013.

[8] http://larbin.sourceforge.net/index-eng.html. This site was last accessed on January 31, 2013.

[9] http://www.phpbb.com/. This site was last accessed on January 31, 2013.

[10] http://drupal.org/. This site was last accessed on January 31, 2013.

[11] http://wordpress.com/. This site was last accessed on January 31, 2013.

nearly the same as the distribution of vulnerability populations in the wild.

The authors also found that most scanners are only effective at following destination links which are mentioned explicitly in the Web pages (e.g., `<ahref=destinationAddress>`). In other words, most scanners were not effective at following implicit links through active content technologies such as Java applets and Flash.

Another important finding in Bau et al. [25] highlights that most scanners were effective in detecting type 1 or simpler XSS vulnerabilities (in which no invalid data is written to the database), e.g., the average percent of "reflected cross-site scripting" faults detected by scanners are 60%. The scanners were also poor at detecting "stored" vulnerabilities (e.g., the "stored XSS" detection rate was roughly 15%, and no scanner could detect second-order SQL injection vulnerabilities) [25]. In stored XSS vulnerabilities, unsanitised user input is directly written to the database and the test cases are then tested by reading in unsanitised values from the database. First-order SQL injection is an SQL injection that the scanner probes for vulnerabilities by executing an SQL command with unsanitised user input. Similar to stored XSS, a second-order SQL injection means that unsanitised values are actually written to the database and then the test cases are checked by reading in invalid values from the database.

Other advanced XSS vulnerabilities include the usage of non-standard tags and keywords, such as `prompt()` and `<style>` [57]. In comparison, the study [25] found the following detection rates for different type of faults:

- "XSS type 1" were detected with 62.5% success rate,
- "XSS advanced" was detected with 11.25% success rate,
- 20.4% of XCS (Cross Channel Scripting) vulnerabilities were detected,
- 21.4% of "SQL first order" vulnerabilities were detected,
- 32.5% of configuration vulnerabilities were detected, and
- 26.5% of session vulnerabilities were detected.

All the scanners used in the study had been approved for PCI Compliance testing.[12] The different types of SQL injection vulnerabilities and XSS (Cross site scripting) vulnerabilities detection techniques are discussed in more detail in the following subsection.

SecuBat [8] is a black-box vulnerability scanner which tries to crawl through Web sites in order to check for SQL and XSS (Cross-site scripting) validation and find security flaws in a Web site through passing it unsanitised user input. There are three main components of the SecuBat vulnerability scanner, namely, the *crawling*component, the *attack* component and the *analysis* component. These are described in more detail below:

The crawling component: The main purpose of this component is to gather a list of Web sites/Web applications to target by the SecuBat scanner. In order to start with a crawling session, the crawler is seeded with a valid default root (Web) address. This address is used as the starting point and a list of all the pages and Web forms which are accessible from this default Web address are collected. This process can be repeated as many times as desired as there are configurable settings in the SecuBat crawler to control the maximum link depth, maximum number of pages per domain to explore/crawl/collect, the maximum time for which the crawling process should continue and the option of whether or not to collect the external links in any page. In order to improve the crawling efficiency, several concurrent worker threads are run during a particular crawling session. Depending on the performance of the host machine, the bandwidth, the targeted Web servers, usually 10 to 30 threads run concurrently during any crawling session. The major implementation of the crawler is based on existing crawling systems, such as the implementation of *SharpSpider* by Moody and Palomino [58] and WebSpider by David Cruwys.[13]

The attack component: After the crawling phase of SecuBat is completed, the processing of the list of collected Web pages starts. The attack component scans each Web page collected during the scanning phase for Web forms. This is mainly because the unsanitised Web inputs (to detect Web vulnerabilities) are submitted to the different Web forms, and as a result, these Web forms serve as the entry points for different unsanitised inputs. For each form, SecuBat automatically extracts the address mentioned in the *action* field of the forms (i.e., the address to which the unsanitised inputs are sent) along with the *method* field (i.e., `GET` or `POST`). The different form fields and the CGI parameters are also collected. Then, the unsanitised inputs for the various form fields are selected depending on the type of the attack launched (SQL injection, simple XSS, encoded XSS or form-redirecting XSS). Finally, the form contents, with the different fields being set to values chosen, are uploaded to the server specified by the *action* address (using either a `GET` or `POST` request). According to the HTTP protocol, the attacked server responds to such a Web request by sending back a response page via HTTP.

The analysis component: After the attack is launched by SecuBat and a response page is sent back by the Web server via HTTP, the analysis component then parses and interprets the response sent. In order to determine whether the attack was successful, an attack-specific response criteria and various keywords (e.g., "sqlexception", "runtimeexception", "error occurred" and "NullPointerException") are used to calculate a *confidence value*. Usually, the confidence value is chosen such that false positives (i.e., the attack is actually not successful but the confidence value indicates otherwise) are reduced.

SecuBat is implemented in C# using Microsoft's Visual Studio.NET 2003 Integrated Development Environment (IDE). In order to store the list of the Web pages collected from the crawling step and the data used to launch, attacks are stored in a Microsoft SQL Server 2000 database server.

[12] The original URL in [25] is no longer accessible. The following URL (https://www.pcisecuritystandards.org/approved_companies_providers/approved_scanning_vendors.php. This site was last accessed on January 31, 2013.) seems to contain the current listing.

[13] http://www.codeproject.com/Articles/6438/C-VB-Automated-Web Spider-WebRobot. This site was last accessed on January 31, 2013.

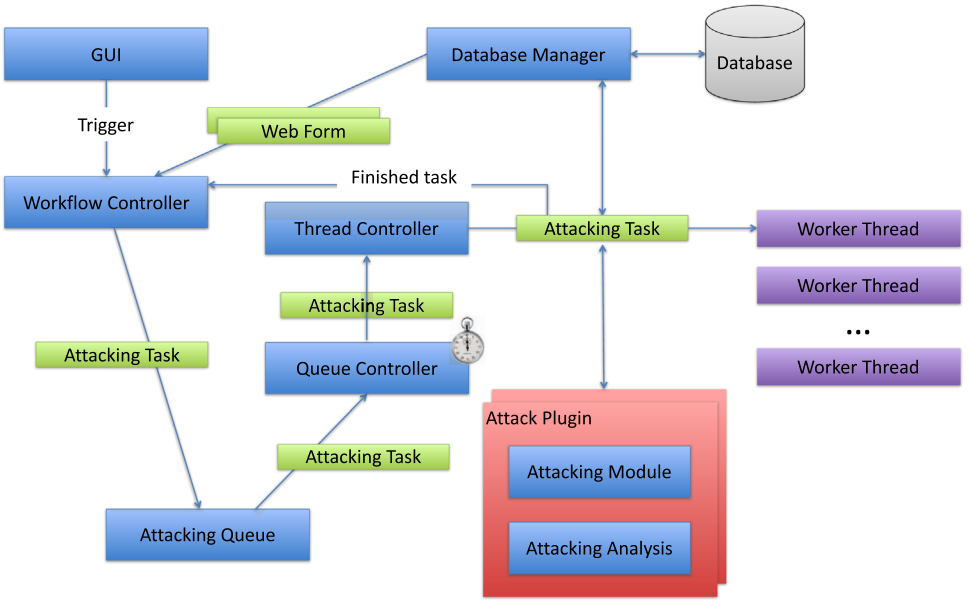

Fig. 6. A diagram showing SecuBat architecture as described in [8, Fig. 3].

Using a DBMS has the advantages of efficient storage and logging of crawling and attack data, custom querying, easy and efficient report generation of crawl and attack sessions and easy access to data collected in previous sessions. The actual architecture of SecuBat is shown in Fig. 6. The tool comprises crawling and attack components, as described earlier and these can be invoked independently. Thus, the architecture allows a user to initiate a crawling session without having to launch an attack, to launch an attack on previously collected historical data from crawling sessions, or to perform a combined crawl and attack step. SecuBat is able to launch about 15–20 parallel attack and response sessions on a typical desktop computer without reaching full load [8].

The tool uses a dedicated *crawling queue*. This queue includes the crawling *tasks* for each Web page, such as target forms and associated links. A *queue controller* checks the crawling queue periodically for new tasks which are then sent to the *thread controller*. The thread controller then chooses a free worker thread, which then executes a task. The *workflow controller* is notified of the discovered links and forms in a Web page by a thread when it has finished executing a task. New crawling tasks are then generated by the workflow controller. Similarly, attack tasks which comprise the attack data to be inserted into the Web pages are stored in a separate queue, known as the *attacking queue*. The queue controller processes the tasks in the queue and assigns them to the worker threads which are available. This assignment is done via the common thread controller.

6.3. Crawling and testing AJAX applications

AJAX-based Web 2.0 applications rely on stateful asynchronous client/server communication, and client-side runtime manipulation of the Domain Object Model (DOM) tree. This not only makes them fundamentally different from traditional Web applications, but also more error-prone and harder to test. In order to detect a fault, a testing method should meet the following criteria [59,60]:

- Reach the fault-execution statements: These statements cause the fault to executed.
- Trigger the error-creation: This causes the fault execution process to generate an incorrect intermediate state.
- Propagate the error: This helps cause a detectable output error as a result of propagating the incorrect intermediate state to the output.

Compared to traditional Web applications, meeting the reach/trigger/propagate criteria is more difficult for AJAX applications [13]. The general approach in testing Web applications has been to request a response from the server (via a hypertext link) and to analyse the resulting HTML. This testing approach based on the page-sequence paradigm has serious limitations meeting even the first reach condition on AJAX sites. Recent tools such as Selenium [61] use a capture-replay style for testing AJAX applications. However, a substantial amount of manual effort is required for testing.

Benedikt et al. [26] present VeriWeb, a tool for automatically exploring paths of multi-page Web sites through a crawler and detector for abnormalities such as navigation and page errors (which are configurable through plugins). In contrast to traditional tools of this sort (e.g., spiders) that usually only explore static links, VeriWeb can also automatically explore the dynamic contents of the Web site, including form submission and execution of client-side scripts [26]. VeriWeb uses *SmartProfiles* to extract candidate input values for form-based pages. Broadly, user-specified SmartProfiles are sets of pairs of attributes and the value for attributes. These attribute-value pairs are then used to automatically populate forms. The specification of the

SmartProfile is independent of the structure of the Web site being tested. Although VeriWeb's crawling algorithm has some support for client-side scripting execution, it is not clear if it would be able to be used for testing in AJAX applications.

The server side of AJAX applications can be tested with any conventional testing technique. On the client side, testing can be performed at different levels. Unit testing tools such as JsUnit can be used to test JavaScript on a functional level. The most popular AJAX testing tools are currently capture-replay tools such as Selenium,[14] Sahi,[15] and Watir,[16] which allow DOM-based testing by capturing events resulting from user interaction. Such tools have access to the DOM, and can assert expected UI behaviour defined by the tester and replay the events.

However, significant manual effort is required on the part of the tester. These include *reaching* the fault in the Web application automatically. Another challenge includes the fact that faulty inputs in AJAX applications can be triggered by various UI events. Thus it becomes important to detect the data entry points in an AJAX applications. In AJAX applications, the data entry points are usually forms in the DOM tree. Also, a faulty state can be triggered by incorrect *sequences* of event executions. Thus, there is a need to generate and execute different event sequences. Also, in AJAX, responses to any client-side event are injected in the single-page interface and faults propagated to the DOM level. These faults are also manifested at the DOM level. Thus, in order to analyse and detect the propagated errors, access to the dynamic runtime DOM is required.

Mesbah and van Deursen propose [13] to use the JavaScript and AJAX crawler Crawljax [62,63] to infer a model of the navigational paths and states by crawling through different UI states and exercising all user interface events of an AJAX site. Crawljax is capable of detecting and firing events on clickable elements on the Web interface. Crawljax can access client-side code and identify clickable elements that result in a state change within the browser's inbuilt DOM. Once the state changes are discovered, a *state-flow graph* is created, which comprises the states of the user interface and the (event-based) transitions which may exist between them. A UI state change in an AJAX application is defined as a change in the DOM tree structure caused either by server-side state changes or client-side state changes. The paths to these DOM changes are also recorded.

Once the different dynamic states have been discovered, the user interface is checked against different constraints. These constraints are expressed as invariants on the DOM tree which allows in checking any state.

Mesbah and van Deursen classify these invariants into three categories based on a fault model, namely DOM-tree invariants, DOM-state invariants and application-specific invariants. The generic DOM-tree invariants are described below.

Validated DOM : this invariant mainly makes sure that there is a valid DOM structure (or valid HTML/JavaScript code) on every possible path of the execution. The DOM tree obtained after each state change is transformed into an HTML instance. A W3C validator acts as an oracle to ensure there are no warnings or errors. This is important because, although most browsers do not give errors as a result of slightly erroneous HTML code, all HTML validators expect that the structure and content is present in the HTML source code. In AJAX applications, however, changes are made on the single-page user interface as a result of partially updating the DOM via JavaScript. Since HTML validators cannot validate client-side JavaScript, this is a problem.

Error messages in DOM : this invariant ensures that the states never contain a string pattern which is the result of an error message. Error messages should be detected automatically (e.g., client-side error messages such as “404 Bad Request”, “400 Not Found” or server-side error messages such as “500 Internal Server Error” and “MySQL Error”).

Other invariants: these include invariants for other things such as discovering links, placing additional security constraints, and invariants which may result in better accessibility anytime throughout the crawling process, etc.

The DOM state machine invariants are described below.

No dead clickables: this invariant mainly ensures that there should be no “dead” or “broken” physical links in an AJAX application. This is important because any clickable link in an AJAX application may actually change the state by retrieving data from the server through JavaScript in the background, which can also be broken. Such error messages are usually masked by the AJAX engine and no dead links are propagated to the user interface. The dead links/clickables can be detected by listening to the client/server request/response traffic after each event.

Consistent back button: this is one of the more common problems in AJAX Web applications (the existence of a broken back button in the browser). Clicking the back button makes the browser completely exit the application's page. Through crawling, a comparison can be made between the expected state in the graph with the state after the execution of the back button and inconsistencies or errors can be automatically detected.

7. Random testing and assertion-based testing of Web services

Random testing of Web applications is a simple and well known technique in which the application is tested by providing random inputs to the application. It can be very effective in certain cases, e.g., Miller, Cooksey and Moore [64] used random testing to detect errors in Mac OS applications, including 135 command-line Unix utilities and 30 graphical applications. They showed that of the 135 command-line utilities which were tested, only 7% of them crashed or hung, which was much better than previous studies. However, it is also well-known [65] that in a lot of cases of random testing, the code coverage is low. This is due to the fact there is a very small probability that all random numbers will be generated correctly to explore all branches of the code. For example, if there is a code block `if (x==10) then..`, and `x` is an integer, then there is only $1/2^{32} \approx 2.33 * 10^{-10}$ chance that the value of `x`

[14] http://seleniumhq.org/. This site was last accessed on January 31, 2013.
[15] http://sahi.co.in/. This site was last accessed on November 5, 2013.
[16] http://watir.com/. This site was last accessed on November 5, 2013.

generated randomly is 10. The ineffectiveness of random testing in terms of coverage is also demonstrated elsewhere [66], where a code coverage of only 39% is obtained by using this technique. In other words, random testing does not perform exhaustive testing. However, random testing can be useful if the software under test is complicated that normal testing would require lots of resources, and when a low to moderate coverage is not a concern. It can also be useful when special value conditions, boundary conditions or invalid conditions need to be checked.

Assertion-based testing is a technique using assertions as an oracle to check whether the test case has been successful or not. Assertions are very commonly used as an oracle to check whether the application behaves correctly or not (e.g., checking expected value against the actual value received). In the following two subsections, we discuss the application of random testing and assertion based testing to Web services. In Section 7.3, we present Artemis, a random testing frame for JavaScript applications.

7.1. Jambition: random testing to Web service

The usage of Web Services has been growing rapidly during the last decade, uncovering new business possibilities. These Web services have also had a very broad and far-reaching influence on our daily life. In addition, the proliferation of Third Generation (3G) and later mobile devices reinforces such growth and leads to new business requirements which takes into account user mobility and connectivity. As a consequence, the Web Service paradigm has to evolve to cope with emerging issues such as:

- more users directly connected and directly interacting with Web Services,
- users connected from any place to each other, and
- increasing levels of complexities for growing business opportunities.

The validation of Web Services is a complex issue and testing solutions must be provided in order to deal with the emerging complexity. For instance, several Web Services involve logical dependencies between their operations. These operations cannot be invoked independently or with any particular order. In addition, a Web service may be a *stateful service*, which means that the results of a particular operation may depend on the data from the previously executed operations of the service. Similarly, the service logic may be dependent on user inputs. As a result, validating these Web services requires developing complicated and thorough test cases. In addition, these test cases should take into account operation dependencies, the states of the service, and data to simulate user inputs. As a result, if test cases were generated automatically, the overall effort required to create a test suite would be greatly lowered, primarily because rigorous validation of Web services does not need to be done by following the detailed program specifications. However, in order to ensure that a set of test cases achieve adequate validation coverage, testing should rely on detailed behaviour specification models [28].

Frantzen et al. [28] propose a tool called *Jambition*, which is a Java tool developed to automatically test Web Services based on functional specifications. In addition, the testing approach adopted by Jambition is random and *on-the-fly*. This means an input is chosen randomly from a given set of inputs, which is then passed to the service and some operation is invoked. The returned message is then retrieved from the service. If the returned message is not allowed by the formal specification, an error is reported, otherwise the next input is chosen and the testing continues.

The on-the-fly approach of Jambition differs from more classical testing techniques by not following the normal practice of generating a set of test cases beforehand which are then executed on the system. On the contrary, test case generation, test case execution, and test case assessment happen in lockstep. As a result, this approach reduces the state space explosion problem faced by several conventional model-based testing techniques. This is because if a tester generates a test case beforehand, the tester needs to test for *all* possible combinations of inputs and outputs returned. However, if a tester is generating and developing test cases on-the-fly, then the tester deals with specific observed output and hence can develop test cases more appropriately. Jambition was tested on the *eHealth Alarm Dispatcher service* [28], and several types of errors were detected during the specification importation and service validation stages. The errors found during the specification importation stage are described below:

- consistency errors in the models, such as a data type being incorrectly referenced in the specification,
- consistency errors between the models and the deployed service: for instance, a parameter of a service operation was declared with different types in the deployed service and in the model, and
- incomplete service deployment problems.

The errors discovered by Jambition during service validation stage are:

- violation of transitional guards: this means that, for example, an emergency condition was not considered fatal even if the functional specification mentioned it as fatal,
- Infinitely running operations or invalid operations, this means that some operations would never terminate due to some defect or no message would be returned by the service or the entire operation would be stopped, and
- unreachable states and transitions: this means that some states or transitions could not ever be reached either due to the faults within the model itself or missing features in the service.

7.2. Testing Web services choreography through assertions

Another approach to test Web services has been proposed by Zhou et al. [27] for specifications written in the Web Service Choreography Description Language

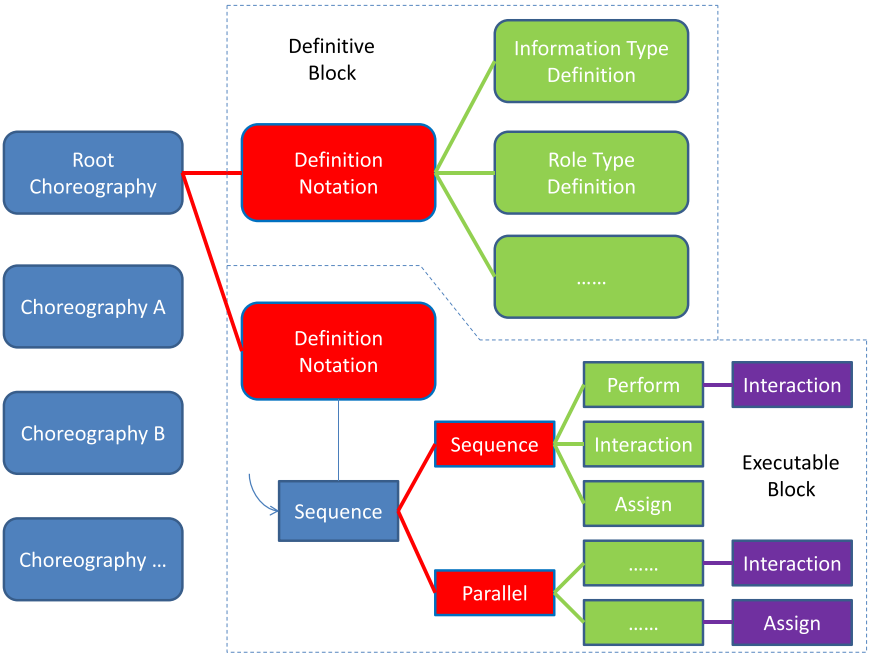

Fig. 7. A diagram showing the program structure of a WS-CDL program as described in [27, Fig. 1].

(WS-CDL),[17] a W3C recommendation for Web Services choreograph. The structure of a WS-CDL program is shown diagrammatically in Fig. 7. The root choreography is the entrance of the WS-CDL program, which is defined uniquely. The specific logic for each choreography is depicted by activity notation. Both the entrance and the behaviour of the choreographies are fixed.

The design and implementation of Web services-based systems require the combination of different distributed services together to achieve the business goal. Thus, it is essential to logically analyse and compose the complex behaviours of Web services. Service Oriented Architecture (SOA) meets this requirement by defining the Web Services compositional languages such as Business Process modelling Language (BPML[18]) and WS-BPEL[19]. There are two models for Web services representation, namely, orchestration model and choreography model. The orchestration model gives a *local* view from a particular business part to handle interactions which allows a particular Web service to perform various internal activities and communicate with other services. The choreography model gives a *global* view regarding collaboration amongst a collection of Web services. Typically such services involve multiple different organisations or independent processes.

WS-CDL gives a global view on the collaboration among a collection of services having multiple participants or organisations. WS-CDL is not an executable language, thus testing it is harder, but testing for Web service choreographies is an effective mechanism to ensure the qualities of system designs [27]. The automated testing approach described by Zhou et al. [27] is very similar to concolic testing approach for testing Web applications (discussed in Section 9). Assertions are used as an oracle in this approach. The dynamic symbolic execution technique is applied to generate test inputs. Assertions are used as test oracles. During the process of symbolic execution, a simulation engine for WS-CDL is used to perform the execution of WS-CDL programs. At the end of each execution, path constraints, which are generated as a result of symbolic execution, are put into a SMT (Satisfiability Modulo Theories) solver. These constraints are then solved to generate new input data and then further constraints are generated in the next simulation. The SMT solver is then used to test whether the assertion predicates evaluate to true under current path conditions for the test data, which further improves the quality of testing.

The analysis of WS-CDL programs is important for the development of Web Services systems. It is also important that the defects are removed as early as possible from these systems, as early defect detection improves quality and cost-effectiveness of the systems. The testing approach proposed by Zhou et al. [27] is shown in Fig. 8.

A WS-CDL program is firstly processed by a custom parser. Since there is an entrance unit for a WS-CDL program, a random value is used initially to drive the choreography. The simulation program performs the dual tasks of simulating the program behaviours and recording the choreography state. Additionally, at the end of every simulation, the symbolic values are analysed by means of symbolic execution. The different predicates which appear in different branches in any simulation are collected together and these predicates then form a Path Constraint (PC). The SMT solver Z3 [67] is used. Z3 solves the path

[17] http://www.w3.org/TR/ws-cdl-10/. This site was last accessed on January 31, 2013.
[18] http://www.ebpml.org/bpml.htm. This site was last accessed on January 31, 2013.
[19] http://www.ibm.com/developerworks/library/specification/ws-bpel/. This site was last accessed on January 31, 2013.

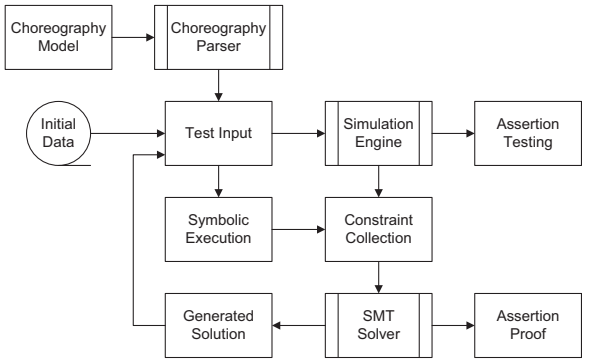

Fig. 8. A diagram showing the testing approach for a WS-CDL program as described in [27, Fig. 2].

constraint by negating the last term of the current path constraint, and the process is continued until a solution for the variables in the path constraint is obtained. The solution obtained is taken as new input for service choreographies to process the next simulation. The testing process finishes when all the branches in the choreography have been traversed by the simulation engine.

The assertions are designed to test whether the current program path or statement satisfies some property. Two methods were designed to generate assertions [27]. The first method uses the current generated test/input data to check whether an assertion is satisfied. However, this method only tests a given assertion using the current data, thus, even if the current data makes the assertion true, there may be some other data which will make the assertion false. The second method enhances the assertion testing where symbolic execution is used along with a constraint solver to achieve a proof that can decide whether the assertion is satisfied under the execution of current path. This process is continued for all the paths for which a given assertion can be tested. When the assertion is satisfied for all the paths in the program, then the assertion is proved to be true for service choreographies. This technique is similar to the one used for bounded model checking proposed in [68]. The main goal in Bounded Model Checking is to search for a counterexample in a certain number of executions. The length of the number of executions has an upper bound of some integer, k. If no bug/error is found in k executions, then the value of k is increased progressively until the testing technique can detect the presence of a bug, or the program becomes intractable, or some pre-known upper bound, known as the *Completeness Threshold* is reached. An experiment was also performed by Zhou et al. [27] for five WS-CDL programs to mainly test path coverage and the number of violated assertions. It was found out that 20% of the assertions resulted in violations and the path coverage for feasible paths was roughly 75%.

7.3. Artemis: a feedback-directed random testing framework for JavaScript applications

In a typical Web application development environment, test frameworks such as Selenium, Sahi and Watir enable capture-replay style testing of Web applications. However, test cases still need to be manually constructed by developers and testers. Given JavaScript's dynamic event-driven nature, it may be difficult to achieve a high coverage and construct test cases that exercise a particular path through execution.

To alleviate the burden of manual test case construction and achieve a high code coverage, Artzi et al. propose Artemis [29], a random testing framework for guided, automated test case generation. The Artemis framework consists of instantiations of three types of main components: (1) an *execution unit* that models the browser and the server, (2) an *input generator* that generates new input sequences, and (3) a *prioritiser* that provides feedback to guide the exploration of the JavaScript application's state space.

The Artemis framework starts at an initial seed URL and iteratively generates, prioritises and executes test inputs. The execution unit triggers events using test inputs, maintains coverage information and records program state and test failures and exceptions. The prioritiser computes the priority of all test inputs and reorders them accordingly. If the program state is not yet visited, the input generator then generates new test inputs in one of three ways:

- it can create an equal-length event sequence by modifying the last event,
- it can produce a longer event by extending the current test input with a new event, or
- it can start executing at a different URL.

The generation of new inputs is guided by feedback provided by the prioritisation strategies and input generation strategies. Artemis define three prioritisation strategies. The default strategy assigns the same priority to all event sequences. The coverage-based strategy assigns priority to event sequences based on the product of coverage of all events in the sequence. The third strategy priorities event sequences by the proportion of written and read variable and parameters in a sequence.

Artemis defines two input generation strategies. The default strategy chooses reasonable default values for parameters. The advanced strategy keeps track of constants while an event is being executed. Such constants are then used as new test inputs. Such a strategy may be more targeted than the default strategy as new inputs are drawn from constants, which have already appeared during execution.

Based on the above prioritisation and input generation strategies, four feedback-directed input generation algorithms are defined. These include the simple algorithm *events* that uses the default strategies, to the most sophisticated algorithm, *all*, which uses all three prioritisation strategies and the advanced input generation strategy.

An experiment with 10 JavaScript programs is conducted to evaluate the effectiveness of Artemis. The evaluation shows that the feedback-directed algorithms (1) improve code coverage significantly over the baseline, which is not feedback-directed and (2) uncover significantly more errors in the programs under test.

The experiment also demonstrates the efficiency of the Artemis framework. As Artemis does not employ sophisticated, expensive constraint-solving algorithms (like that used in Kudzu [16], discussed in Section 9.4), it only takes at most 2 min to generate 100 test cases with all the input generation algorithms. In comparison, a 6-h timeout is applied for Kudzu.

7.4. JSContest: contract-driven random testing of JavaScript

Heidegger and Thiemann propose JSConTest [30], a random testing framework, that includes a contract language for JavaScript that allows a programmer to annotate a JavaScript program with functional contracts. The language supports the following contracts.

- The traditional type-based contracts, such as `/** (object) → bool */`.
- Composite contracts that are built from primitive ones and enriched by static analysis information. Such information, including `@numbers`, `@strings` and `@labels`, can be used as guide to collect information from the annotated function.
- Dependencies that may exist among function parameters. Such dependencies can be used to more efficiently find a counterexample.
- Annotations that control the generation of assertions and tests. These include `~noAsserts`, which specifies that no assertions should be generated for a given contract, `~noTests`, which specifies that no contracts should be added to the test suite, and `#Tests:`*i*, which specifies the number of tests to be generated for a given contract.

A simple JavaScript program with two contracts is shown in Fig. 9.

JSContest relies on annotations such as `@numbers` to guide the generation of random tests in order to improve the possibility of finding a counterexample.

JSConTest also includes a runtime monitoring tool for contracts that insert assertions into function bodies and check whether function execution is successful or not.

The framework is evaluated on a custom JavaScript implementation of a Huffman decoder. Mutation testing techniques are employed to generate mutant programs with a small number of mutation operators. Out of the 716 generated mutants, about 88% are rejected by the JSContest test suite. Such a high mutant killing percentage suggests that JSContest is effective at detecting type errors.

8. Fuzz testing

Fuzz testing is an effective technique for finding security vulnerabilities in software by testing the application with boundary values, invalid values or values that are rarely used. Fuzz testing tools create test data by (1) applying random mutations to well-formed inputs of a program or (2) by generating new test data based on models of the input. Fuzz testing can in generally be divided into two categories, namely, white-box fuzz testing and black-box fuzz testing.

The main advantage of fuzz testing is that testing is focussed on using special values as input to the program under test and thus helps it in detecting critical, exploitable bugs which would probably not be detected by model-based approaches. Additionally, the overall approach to testing is quite simple (it is essentially random testing combined with symbolic execution) and complete (i.e., it tries to cover as many branches as possible).

In Section 8.1 we introduce the notion of white-box fuzz testing, and present several techniques to improve its efficiency. In Section 8.2, FLAX, a black-box fuzz testing framework for JavaScript, is presented to demonstrate the application of such techniques in Web application testing.

8.1. White-box fuzz testing

White-box fuzzing [31] techniques combine fuzz testing with dynamic test generation [65,69]. White-box fuzzing executes the program under test with an initial, well-formed input, both concretely and symbolically. During the execution of conditional statements, symbolic execution creates constraints on program inputs. Those constraints capture how the program behaves when fed these inputs, and satisfying assignments for the negation of each constraint define new inputs that exercise different control paths. White-box fuzzing repeats this process for the newly created inputs, with the goal of exercising many different control paths of the program under test and finding bugs as fast as possible using various search heuristics. In practice, the search is usually incomplete due to a large and infeasible number of control paths and because the precision of symbolic execution, constraint generation and solving is inherently limited. However, it

```
1  /** int → int */
2  function f(x) { return 2 * x; };
3
4  /** (int, int) → bool */
5  function p(x, y) {
6    if (x != y) {
7      if (f(x) == x+10) return "true"; // error
8    };
9    return false;
10 };
```

Fig. 9. A simple JavaScript program with contracts [30, Fig. 1].

is a commonly used approach in detecting program vulnerabilities in large applications.

The current effectiveness of white-box fuzzing is limited when testing applications with highly structured inputs, such as compilers and interpreters. The inputs for these applications are processed in stages, such as lexing, parsing and evaluation. Due to the large number of control paths in early processing stages, white-box fuzzing, in the presence of a well-formed input set, rarely proceeds beyond these initial input processing stages. For instance, there are many possible sequences of blank spaces, tabs and carriage returns separating tokens in most structured languages, each of which corresponds to a different control path in the lexer. In addition to path explosion, symbolic execution may fail in early processing stages. For instance, lexers often detect language keywords by comparing their precomputed, hard-coded hash values with the hash values of strings read from the input. This effectively prevents symbolic execution and constraint solving from ever generating input strings that match the keywords since hash functions cannot be inversed (i.e., if we have a constraint `x=hash (y)` and a value of `x` is given, then we cannot compute `y` which satisfies the constraint).

One possible approach to overcoming this problem is proposed in [31], where the algorithm records an actual run of the software under test on a well-formed input, symbolically evaluates the recorded trace and records the different constraints on inputs, which shows how the program behaves under the inputs. The collected constraints are then negated one by one and solved with a constraint solver, producing new inputs that allow the exploration of different control paths in the program. This process is repeated with the help of a code-coverage maximising heuristic designed to find defects as fast as possible. This approach employs dynamic test generation techniques such as DART [65] and EXE [69], which are introduced in Section 9.

For example, for the symbolic execution of the code fragment `if (x==20)`, the initial value of the variable `x` is set to 5, then the execution of the program leads to the constraint `x !=20`. This constraint is then negated and solved, which results in the new constraint `x=20`, which ensures that the `if` statement is executed and further constraints are collected. This allows to exercise and test additional code for security bugs, even without specific knowledge of the input format. Furthermore, this approach automatically discovers and tests corner cases where programmers may fail to properly allocate memory or manipulate buffers, leading to security vulnerabilities.

This algorithm is implemented in the Microsoft whitebox testing tool SAGE (Scalable, Automated, Guided Execution) [31], which is a tool employing x86 instruction-level tracing and emulation for white-box fuzzing of arbitrary file-reading Windows applications.

SAGE was used for testing several windows applications. Without any format-specific knowledge, SAGE detects the MS07-017 ANI vulnerability,[20] which was missed by extensive black-box fuzzing and static analysis tools. SAGE has also discovered more than 30 new bugs in large Windows applications including image processors, media players, and file decoders.

Another approach to solve the limited coverage problem due to white-box fuzz testing is proposed by Godefroid et al. [33], which enhances white-box fuzzing when applying it to complex structured-input applications with a grammar-based specification of valid inputs. A novel dynamic test generation algorithm is proposed where symbolic execution directly generates grammar-based constraints whose satisfiability is checked using a custom grammar-based constraint solver. The algorithm is implemented and evaluated on a large security-critical application, the JavaScript interpreter of Internet Explorer 7 (IE7). Results of these experiments show that grammar-based white-box fuzzing explores deeper program paths and avoids dead-ends due to non-parsable inputs. Compared to regular white-box fuzzing, grammar based white-box fuzzing increased coverage of the code generation module of the IE7 JavaScript interpreter from 53% to 81% while using three times fewer tests [33]. Moreover, the grammar-based constraint solver can complete a partial set of token constraints into a fully defined valid input, hence avoiding exploring many possible non-parsable completions. By restricting the search space to valid inputs, grammar-based white-box fuzzing can exercise deeper paths, and focus the search on the harder-to-test, deeper processing stages.

8.2. FLAX: a black-box fuzz testing framework for JavaScript

Saxena et al. [32] have adopted a hybrid technique which is a combination of dynamic taint analysis [70] and fuzz testing. Dynamic taint analysis basically executes a program and then determines which combination of program paths is affected by user inputs or other predefined taint sources. Dynamic taint analysis can also be used to detect inappropriate user input values during a program execution [71]. For example, dynamic taint analysis can be used to prevent attacks caused as a result of an attacker entering malicious lines of code into the user input fields, which may result in the user (of the Web application) inadvertently executing some undesirable code (also known as code injection attacks), by monitoring whether user input is executed [72,71]. However, dynamic taint-tracking alone cannot determine if the application has been sufficiently validated against unsafe data before using it, especially when parsing and validation checks are syntactically indistinguishable. If an analysis tool treated all string operations on the input as parsing constructs, then the validation checks will not be identified and the taint analysis will result in an enhanced number of false positives [32]. Conversely, if the analysis treats any use of unsafe data which has been parsed and/or passed through a validation construct as safe, it will probably miss many bugs, which will result in false negatives.

Saxena et al. [32] use their testing technique to detect client-side validation (CSV) vulnerabilities. These vulnerabilities primarily arise as a result of not doing enough validations on the unsafe data/input used at the client-side (usually written in JavaScript) of Web applications. This testing technique is light-weight, efficient, and have no false positives. Saxena et al. [32] incorporate this technique

[20] http://technet.microsoft.com/en-us/security/bulletin/ms07-017. This site was last accessed on January 31, 2013.

into a prototype, called FLAX. This tool also scales to real-world applications. The dynamic analysis approach proposed in FLAX [32] to discover vulnerabilities in Web applications is called *taint enhanced blackbox fuzzing*. This technique overcomes the above-mentioned limitations of dynamic taint analysis by using random fuzz testing to generate test cases that concretely demonstrate the presence of a CSV vulnerability. This eliminates the problem of false positives and false negatives which would otherwise result from the usage of a purely taint-based analytic tool. In the preliminary study comprising 40 real-world and popular JavaScript-intensive programs, it was found that FLAX was able to discover 11 CSV vulnerabilities in the wild. These JavaScript-based programs included several third-party iGoogle gadgets, Web sites, AJAX applications and third-party libraries. These vulnerabilities were unknown prior to the experiments. These findings also confirmed [32] that CSV vulnerabilities are both conceptual and widespread in Web applications today.

9. Concolic Web application testing

Concolic (concrete and symbolic) testing techniques automate test input generation by combining the concrete and symbolic (concolic) execution of the software under test. Most test input generation techniques use either concrete execution or symbolic execution that builds constraints and is followed by a generation of concrete test inputs from these constraints [73]. Concolic testing, on the other hand, combines both these techniques, which take place simultaneously. The goal in concolic testing is to generate different input data which would ensure that all paths of a sequential program of a given length are covered. The program graphs, which depict program statements and the program execution, are provided as inputs.

In Section 9.1, we introduce concolic execution and testing in general, motivating its effectiveness through a small example, and present a number of well-known concolic testing tools.

PHP is a very popular Web application programming language. It is a scripting, interpreted language that is widely used to create server-side applications. JavaScript is a widely used, client-side scripting language found in virtually all Web applications. Their popularity and dynamic nature make their thorough testing difficult yet highly critical. In Sections 9.2– 9.4, we present three concolic testing approaches that generate tests and defects detection for PHP and JavaScript applications, exploiting string- and path-based constraints, respectively. The benefits of concolic testing, especially on dynamic languages such as PHP and JavaScript, can be evidently demonstrated through case studies.

9.1. Concrete, symbolic execution

Concolic testing uses concrete values as well as symbolic values for input and executes a program both concretely and symbolically, called concolic execution. The concrete part of concolic execution is where the program is normally executed with concrete inputs, drawn from random testing. The symbolic part of concolic execution collects symbolic constraints over the symbolic input values at each branch point encountered along the concrete execution path. At the end of the concolic execution, the algorithm computes a sequence of symbolic constraints corresponding to each branch point. The conjunction of these symbolic constraints is known as *path constraints*. More formally, a path constraint (PC) is defined as the conjunction of conditions on variables as a result of executing the Web application with concrete values. All input values that satisfy a given path constraint will cover the same execution path. Concolic testing first generates random values for primitive inputs and the NULL value for pointer inputs. The algorithm then executes the program concolically in a loop. At the end of execution, a symbolic constraint is negated in the original path constraint (which contains a conjunction of symbolic constraints) and the alternative branches of the program are explored. The algorithm is continued with the newly generated concrete inputs for the new path constraint. As a result, concolic testing combines random testing and symbolic execution, thus overcomes the limitations of random testing, such as the inefficient and ad hoc nature of the test cases generated [64], the difficulty in traversing the different paths in a program, redundant test input data which lead to the same observable program behaviours [74], and the low coverage obtained (due to the random nature of the input data) [65].

In order to obtain a better understanding of how these concolic testing techniques work for a program, we could consider the code block shown in Fig. 10.

Since it is very difficult with random testing or manual testing to generate input values that will drive the program through all of its different execution paths, Concolic testing is better suited to obtaining the desired program coverage because it is able to dynamically gather knowledge about the execution of the program in a *directed* search.

```
1 int f(x) {
2   return 2*x;
3 }
4 int h(x,y) {
5   if (x != y)
6     if (f(x) == x + 10)
7       error();
8 }
```

Fig. 10. A simple pseudo procedural program.

In the above code block, a concolic testing technique assigns random values to both `x` and `y` in the function `h`. Assume that the concrete values assigned to `x` and `y` are `x=0` and `y=1`. With these assignments, the inner `if` statement is not reached (since `f(x)=0` and `x+10 != 10`). Along with the normal concrete execution, the predicates $x_0 \neq y_0$ and $2*x_0 = x_0+10$ are formed on-the-fly according to how the conditionals evaluate. In this case, x_0 and y_0 are symbolic variables which represent the memory locations of the concrete variables `x` and `y`. The expression $2*x_0$, or `f(x)` is defined through an inter-procedural dynamic tracing of symbolic expressions.

Sometimes, the path constraints may be too complicated for a constraint solver to generate concrete values to satisfy the constraint. In such cases, some symbolic constraints are replaced with concrete values. Thus, concolic testing is complete only if a given oracle can solve all constraints in a program, and the length and the number of paths are finite [73]. There are a few concolic testing tools which are commonly used. The most popular ones are EXE [69], DART [65], and CUTE [74].

EXE [69] is an effective bug finding tool that automatically generates inputs with the aim of crashing real code. Instead of running code on manually or randomly constructed input, EXE runs on symbolic inputs, initialised with a random input. EXE tracks the symbolic constraints for each memory location. If a statement uses a symbolic value, EXE does not execute it as a normal statement. Instead the constraint is added as a conjunction to the list of path constraints. If a symbolic execution is checked conditionally by a program, EXE forks execution, and the expression is constrained to be true on the true branch, and false on the other branches. Since EXE takes into account all the possible values along a path, it can force the program to be executed down any feasible program path, and at statements or along program points where dangerous operations (e.g., a pointer dereference) are allowed/performed, it can identify whether the current path constraints would allow any value that causes a defect [69]. When a path terminates or hits a bug, EXE automatically generates a test case by solving the current path constraints to find concrete values using its constraint solver, the Simple Theorem Prover (STP) [75]. For an `assert` statement, EXE can reason about all possible input values on the given path that may cause the assert to fail. If the assert does not fail, then either, (1) no input on this path can cause it to fail, (2) EXE does not have the full set of constraints or (3) there is a bug in EXE.

The ability to automatically generate concrete inputs to execute program paths has several features. First, EXE can test all paths/branches of a program exhaustively (given enough time), which is impossible to do in cases of manual or random testing. Second, EXE generates actual attacks. Third, the presence of a concrete input allows the user to easily discard error reports due to bugs in EXE or STP: the user can confirm the error report by simply rerunning an uninstrumented copy of the checked code on the concrete input to verify that it actually hits the bug (both EXE and STP are sound with respect to the test cases generated, and therefore false positives can only arise due to implementation bugs in EXE and/or STP).

Proper customisation makes STP often 100 times faster than more traditional constraint solvers while handling a broader class of examples. Crucially, STP efficiently reasons about constraints that refer to memory using symbolic pointer expressions, which presents a few challenges. For example, given a concrete pointer `a` and a symbolic variable `i` with the constraint $0 \leq i \leq n$, the conditional expression "`if (a[i]==10)`" is essentially equivalent to testing all the different values of `a[i]`, where `i` is between 1 and 10. Thus, we are essentially testing a big disjunction of different possible conditions which may be true: "`if(a[0]==10||...||a[n]==10)`". Similarly, an assignment `a[i]=42` represents a potential assignment to any element in the array between `0` and `n`.

Directed Automated Random Testing (DART) [65], similar to EXE [69], is used for automatically testing software and is composed of three main techniques, (1) automated extraction of the interface of a program with its external environment using static source-code parsing, (2) automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in and (3) dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths.

CUTE [74] stands for Concolic Unit Testing Engine. CUTE implements a solver so that the test inputs are generated incrementally for both arithmetic and pointer constraints. This solver performs several optimisations [74], which help to improve the testing time by several orders of magnitude. The experimental results demonstrated in [74] confirm that CUTE can efficiently explore branches in a C program and achieve high branch coverage. CUTE was also shown to be efficient at detecting software bugs that result in assertion violations, segmentation faults, or infinite loops [74].

The methodology used in CUTE that makes it efficient to test programs written in C efficiently is based on separating pointer constraints from the simpler arithmetic constraints. Additionally, in order to make the entire symbolic execution *light-weight* and ensure that the constraint solving procedure is tractable, pointer constraints are kept simple. The logical map, which keeps tracks of the different pointer relations, is used to simplify and replace complex symbolic pointer constraints with simpler symbolic pointer constraints. For example, if p is an input pointer to a `struct` with a `field` f, then a constraint on $p \rightarrow f$ will be reduced to a constraint on f_0 , where f_0 is the symbolic value representing the constraint $p \rightarrow f$. Although this over-simplification of complex symbolic pointer expressions may give rise to some approximations that do not precisely capture all executions, the original relationship between the different pointers (as defined in the logical map) is still maintained, and the approximations are also sufficient for testing purposes. Furthermore, this simplification of pointer constraints results in pointer constraints of the form $x = y$ or $x \neq y$, where x and y are either symbolic pointer variables or the special C constant `NULL`. The advantage of reducing complex pointer constraints to simpler constraints of the above-mentioned form is that such simple constraints can be solved efficiently [74].

9.2. A string-based concolic testing approach for PHP applications

Wassermann et al. [35] develop a concolic testing-based framework for detecting SQL injection defects through dynamic test input generation by manipulating string operations in PHP. String operations are modelled using *finite state transducers*. Constraints are then used to generate new string values. An algorithm is then used to validate whether a string constitutes an SQL injection attack. Finally, *backward slices* are dynamically constructed at runtime to enable testing beyond unit level.

Scripting languages such as PHP support meta-programming capabilities and hence are more dynamic. For instance, PHP allows function call names and variable names to be constructed dynamically, from user inputs. The presence of such dynamic language features makes it very hard for static analysis tools to analyse PHP programs. The approach [35] tackles this challenge by using a concolic approach that records variable values in concrete execution and use them as constraints to generate new inputs symbolically.

For example, the PHP program in Fig. 11 contains an unsafe database call: on line 18 the value of variable `userid` is used without sanitisation, making it vulnerable to SQL injection attacks.

In this framework, new string-typed test inputs are generated from current string values through the use of constraints. The generation of such constraints is enabled by modelling string operations and type casts as *finite state transducers* (FSTs), finite state machines with an input tape and an output tape. As in concolic testing, constraints are generated, solved and inverted to generate new test inputs to cover different paths of the PHP program under test.

Focused on detecting SQL injection vulnerabilities, this framework makes use of Grasp [76], a modified PHP interpreter, as test oracle. Grasp performs character-level tainting, allowing security policies, in this case those related to SQL injections, to be defined. Symbolic execution and FSTs are used in combination to record and generate new query strings that are potentially attacks.

The state space of any non-trivial program may be too large for a concolic testing technique to handle efficiently. Sometimes, significant portions of a program's execution, such as logging, are not relevant to the properties of interest. This problem is alleviated [35] by analysing program points that are (directly or indirectly) relevant to possible failure, in a backward manner. Starting at function calls that send a query to the database, other functions where this call occurs are iteratively added. Control dependency and data dependency are resolved by maintaining a stack trace of function calls, and by examining symbolic values during execution. Approximately this process constructs a backward program slice, and is shown to dramatically reduce the number of constraints generated, sometimes by several orders of magnitude.

The proposed framework is implemented and evaluated on three PHP Web applications with known SQL injection vulnerabilities: Mantis 1.0.0rc2, Mambo 4.5.3 and Utopia News Pro 1.3.0. The concolic testing framework manages to detect SQL injection vulnerabilities in all three applications using at most 23 inputs. In the case of Mantis and Mambo, as few as 4 and 5 inputs are required respectively to detect a vulnerability.

```
1  isset ($_GET [ 'userid' ]) ?
2    $userid = $_GET [ 'userid' ] : $userid= '' ;
3  if ($USER[ 'groupid' ]!=1) {
4    // permission denied
5    unp_msg($gp_permserror );
6    exit;
7  }
8  if ($userid == '' ) {
9    unp_msg($gp_invalidrequest );
10   exit;
11 }
12 $userid = "00".$userid;
13 if (! eregi('00[0-9]+', $userid)) {
14   unp_msg('You entered an invalid user ID.' );
15   exit;
16 }
17 $getuser = $DB->query("SELECT * FROM"
18   . "'unp user' WHERE userid='$userid'" );
19 if (!$DB->is_single_row($getuser)) {
20   unp_msg('You entered an invalid user ID.' );
21   exit;
22 }
```

Fig. 11. An example PHP program from [35, Fig. 1] that contains an SQL injection vulnerability.

9.3. Apollo: a path-based concolic testing framework for PHP applications

Artzi et al. [34] propose a concolic testing technique for PHP Web applications. The path constraints which are generated from symbolic execution are stored in a queue. The queue is initialised with the empty path constraint. A constraint solver is then used to find a concrete input that satisfies a path constraint taken from the queue. The program is executed *concretely* on the input and tested for failures. The path constraint and input for each detected failure are merged into the corresponding bug report. Now the program is executed *symbolically* on the same concrete input value (chosen by the constraint solver) and different path constraints are obtained (i.e., a boolean expression with a conjunction of conditions which are true when the application is executed with an input). New test inputs are then created by solving modified versions of the path constraint obtained. If a solution exists to an alternative path constraint which corresponds to an input, then the execution of the program will be completed along the opposite branch [34].

In order to obtain the different modified versions of the path constraint, the last condition in the original path constraint is removed and the last conjunct in the new path constraint is negated. For example, assuming the original path constraint obtained from initial concrete input is the following:

$x \wedge y \wedge z$

Then, the additional path constraints are

$x \wedge \neg y \wedge \neg z, \quad x \wedge \neg y \quad \text{and} \quad \neg x$

The PHP program in Fig. 12 illustrates how Apollo can be applied to testing PHP applications.

The program execution starts with the empty input, with the variable `page` being set to the initial value 0 after line 3 of the program execution. Since the variable `login` is not defined, the function `validateLogin()` (not shown here) is not invoked, generating the following path constraint (*PC*), where *NotSet*(`page`) is due to the conditional statement in line 3 being executed:

$PC : \mathit{NotSet}(\texttt{page}) \wedge \texttt{page} = 0 \wedge \texttt{page2} \neq 1337 \wedge \texttt{login} \neq 1$

This original path constraint is then updated by removing the last constraint and negating the last constraint from this updated path constraint. Thus, in this case, 4 new path constraints are generated (*PC*1 to *PC*4).

$PC1 : \mathit{NotSet}(\texttt{page}) \wedge \texttt{page} = 0 \wedge \texttt{page2} \neq 1337 \wedge \texttt{login} = 1$

$PC2 : \mathit{NotSet}(\texttt{page}) \wedge \texttt{page} = 0 \wedge \texttt{page2} = 1337$

$PC3 : \mathit{NotSet}(\texttt{page}) \wedge \texttt{page} \neq 0$

$PC4 : \mathit{Set}(\texttt{page})$

Similarly, these constraints are solved by the constraint solver that enables exploration of different paths of the program execution. For example, for the path constraint *PC*3, the constraint solver may assign any value to "`page`" other than 0.

```
1  <?php
2    make_header(); // print HTML header
3    if(!isset($_GET["page"])) $page = 0;
4    else $page = $_GET["page"];
5    if($_GET["page2"] == 1337) {
6      require("printReportCards.php"); //a separate PHP
           application
7      die(); // terminate the PHP program
8    }
9
10   if($_GET["login"] == 1 ) validateLogin();
11
12   switch ($page) {
13     case 0: require("login.php"); break;
14     case 1: require("TeacherMain.php"); break;
15     case 2: require("StudentMain.php"); break;
16     default: die("Incorrect page number. Please verify
           .");
17   }
18
19   make_footer(); // print HTML footer
20   ...
21 ?>
```

Fig. 12. A simple PHP program from [34, Fig. 1] that contains some defects.

9.3.1. Minimisation of path constraints

A defect report, or a *bug* report contains a set of path constraints leading to inputs exposing the failure. A lot of dynamic test generation tools [65,74,69] present the entire input to the user without an indication of which subset of the input is responsible for the failure. In order to remove the irrelevant constraints, path constraints can be minimised [34].

In order to do this, the intersection of the different path constraints exposing a failure is taken for a given defect report. Iteratively, each condition from the intersection of the path constraints is removed one by one. If the shorter path constraint (after removing the condition) does not expose the failure, the constraint is added back as it is required for exposing the failure. From the minimised path constraint, the algorithm produces a concrete input that exposes the failure. Although the algorithm does not guarantee that the algorithm returns the *shortest* path constraint necessary to expose the failure, the algorithm is simple, fast and effective.

For example, for the two path constraints PC_a and PC_b, which expose the same failure and the constraints are:

$$PC_a : x \wedge y \wedge z \wedge a$$

$$PC_b : \neg x \wedge y \wedge z$$

Taking the intersection of these two path constraints, the new path constraint PC_{new} is $PC_{new} : y \wedge z$. The algorithm takes out conditions from the constraint one by one. If the remaining constraint exposes the failure, that condition is removed and PC_{new} is updated, otherwise it is kept. The algorithm terminates when there are no more conditions which can be removed.

9.3.2. Implementation technique

Artzi et al. [34] also proposed a tool to implement the technique proposed by them. The architecture of the framework is shown in Fig. 13. As shown in the diagram, the framework Apollo comprises three major components, the *Executor*, the *Input Generator* and the *Bug Finder*. The *Executor* carries out the testing of a particular PHP file(s) with a given input. The *Executor* ensures that the database is in an appropriate state before each cycle of an execution. The *Executor* has two sub-components, namely:

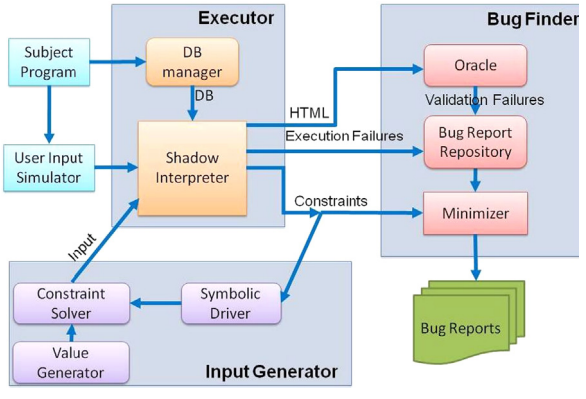

Fig. 13. The architecture of Apollo as shown in [34, Fig. 4].

- *The Database Manager*: This component, as the name suggests, is responsible for initialising a database for use during the execution stage by the PHP application. The state of the database is also restored after each execution.
- *The Shadow Interpreter*: This component is mainly a modified PHP interpreter responsible for storing the different path constraints (PC) generated throughout the program execution. This component also stores different positional information regarding the input.

Similarly, the *Bug Finder* uses an oracle to find HTML failures, keeps a track of the different bug reports and minimises the path constraints which are responsible for a particular fault. The *Bug Finder* mainly comprises three components, which are:

- *The Oracle*: The oracle is basically a heuristic which is used to find defects in any given program. In the case of PHP applications, the oracle is designed to detect HTML failures (such as invalid HTML syntax) in the program output.
- *The Bug Report Repository*: This is responsible for storing all the bug reports containing a detailed description of the different HTML failures and execution failures found during program execution.
- *Input Minimiser*: The *Input Minimiser* takes a given bug report as the input and finds the smallest set of path constraints on the input parameters required to produce the same (type of) faults as those described in the bug report.

The *Input Generator* contains the main implementation of the algorithm for obtaining a collection of path constraints and minimising them. The *Input Generator* comprises the following three sub-components:

- *The Symbolic Driver*: This component generates new path constraints and also selects a path constraint (according to the state of the priority queue) to be solved for a given execution.
- *The Constraint Solver*: This is mainly used to compute the assignment of values to input parameters that satisfies a given path constraint (i.e., it is used to find solutions to a path constraint).
- *The Value Generator*: This component generates values for input parameters that are not constrained. The values are either randomly generated or constant values extracted from the program source code.

This technique [34] gives a practical example of how concolic testing can be applied to testing Web applications. The algorithm used for minimisation of path constraints also helps minimise the time taken for the testing process. Additionally, the testing process is complete as the testing process continues until all the branches are covered.

9.4. Kudzu: a symbolic testing framework for JavaScript

JavaScript [6] is the lingua franca for client-side scripting in Web applications. It is a powerful language with many advanced features, including dynamic typing and evaluation, functions as objects, and various forms of delegation. Such features enable sophisticated, interactive Web applications to be created using JavaScript. These features also make it very challenging to thoroughly test a JavaScript application. For instance, a JavaScript application may accept many kinds of inputs, including responses from the server and user input from fields, which are structured as strings. A JavaScript testing tool must then be able to discern the different kinds of inputs and handle them accordingly.

Kudzu [16] developed by Saxena et al. is a symbolic testing framework for JavaScript with the aim of detecting client-side code injection vulnerabilities. Kudzu groups JavaScript inputs into two categories: (1) event space that encompasses states and sequence of actions of UI elements, and (2) value space that encompasses values supplied by external entities, including form data from users, HTTP requests and URLs. Kudzu produces high-coverage test cases systematically by generating constraints about such inputs in its own constraint language, which supports Boolean, bit vectors and string constraints. A practical constraint solver, Kaluza, is developed as part of Kudzu to solve such constraints.

The overall architecture of Kudzu can be seen in Fig. 14. The core components of the system are those shaded in gray.

- The *GUI explorer* selects a random ordering of user events and executes them. Concrete inputs of an execution is recorded, and then symbolically executed by the *dynamic symbolic interpreter*.
- The *path constraint extractor* takes the results of symbolic execution and constructs constraints with the aim to exercise different execution paths of the JavaScript code. Kudzu uses a generational search strategy proposed in [31] (introduced in Section 8.1) to decide the order of exercising branches.
- The *constraint solver* solves the constraints by finding satisfying assignments to variables, therefore generating new values to be used as inputs. Finally, the *input feedback system* sends the newly generated inputs back to the JavaScript program to drive new execution.

The other three components in Fig. 14 are application-specific. Specifically,

- The *sink–source identification* components identify critical sinks that may receive untrusted inputs using a data flow analysis technique.
- The *vulnerability condition extractor* combines formulae generated by the symbolic interpreter and path constraints to create formulae that describe input values to the critical sink.
- Finally, the *attack verification* component executes tests with the generated inputs and determines whether an attack is executed.

The Kudzu system is evaluated on 18 real-world JavaScript applications, which are used to evaluate the authors' previous system, FLAX [32] (discussed in Section 8.1). Compared to FLAX, Kudzu has a number of advantages. Firstly, Kudzu requires a test suite to reach vulnerable code, whereas Kudzu automatically generates test inputs that lead to high code coverage. Secondly, FLAX only performs black-box fuzzing, a form of random testing, whereas Kudzu, based on concolic execution, is able to reason about possible vulnerabilities, hence is more directed. As a result, Kudzu manages to find code injection vulnerabilities in 11 of the 18 applications. Among the vulnerabilities discovered by Kudzu 2 were not known prior to the experiments and are missed by FLAX.

10. User session-based testing

The white-box testing approach is often limited in nature because the inputs have to be selected manually in order to adequately test the different parts of the system. The selection of such inputs is a complicated process and may take a long time. A user session starts when a request from a new IP address is received by the server and ends when the user leaves the Web site or the session times out [36,9]. User session-based testing techniques help alleviate both of the above problems. In user session-based testing, a list of interactions performed by the user in the form of URLs and name-value pairs (of different attributes), or cookies, are collected, from which test cases are then generated.

The rationale and advantages of user session-based testing are based on the fact that since most Web application

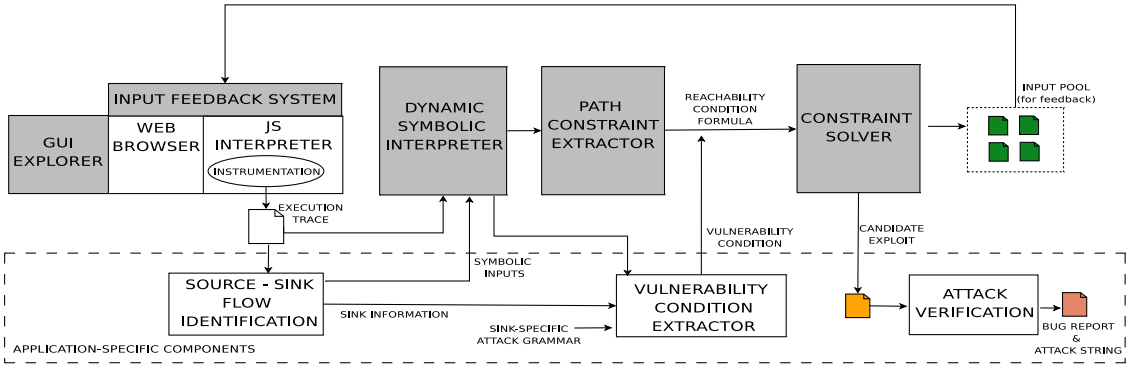

Fig. 14. The overall architecture of Kudzu as shown in [16, Fig. 1].

operations comprise receiving requests from the client and then processing those requests, the collection of client requests can be done with minor tweaks to the server. For example, with minimal configuration changes, the Apache server can log all the received GET requests. Utilising Java servlet filters is yet another alternative that enables the dynamic interception of requests and responses at the server side. Another advantage of user session testing is that since we are only concerned with the sessions' data (URL and name-value pair), it provides a high-level abstraction of the heterogeneous nature of the different components. This lessens the dependencies of user session-based testing techniques on changes in components of the Web application under test.

Assuming we have a set of m user sessions $U=\{\mu_1,\mu_2,\ldots,\mu_m\}$, such that each user session μ_i sends n requests $\{r_1,r_2,\ldots,r_n\}$ to the Web server, where each request r_j consists of a URL and a name-value pair, Elbaum et al. [36] propose three different techniques for generating test cases from user sessions, namely:

- Using user session data directly. In this technique, different user sessions are *replayed* individually. More specifically, each individual request r_j from a user session μ_i is formatted into a HTTP request which is then sent to the server. The resulting test suite thus comprises m test cases for user sessions. Elbaum et al. only include requests that are the result of a sequence of the user's events at the server site.
- Replaying a mixture of different session interactions from different users. This method is supposed to expose the error conditions caused when conflicting data is provided by different users. The test cases are generated, and a new test suite is created from the set of different user sessions, U, according to Algorithm 1.
- Mixing regular user requests with requests that are likely to be problematic (e.g., navigating backward and forward while submitting a form). In this case, the test cases are generated by replaying user session with some modifications. This technique involves randomly deleting characters in the name-value pair strings which may result in different scenarios being explored (e.g., deleting one character from the *username* and *password* in a login form leads to an incorrect test case). The algorithm can be described more formally [36] as demonstrated in Algorithm 2:

Elbaum et al. [36] show that these capture-and-replay techniques for user session-based testing are able to discover certain special types of faults which cannot be exposed by white-box testing. However, their technique cannot detect faults that are a result of rarely entered data. They also show that as the number of user sessions increases, the effectiveness (percentage of faults detected) of these techniques increases. However, as the number of user sessions increases, the time and the cost associated with collecting the data, analysing the data and generating the test cases also increase. Thus, there is a need for prioritising and reducing the number of test cases.

Algorithm 1. The replay algorithm used in user session-based testing where the interactions of different sessions of different users are taken into account [36].

```
Data: The set of m user sessions, U = {μ1, μ2, ..., μm}
Result: The new test suite, Tnew
Tnew ← ∅;                                                    // Initialisation
while there are unused sessions in U do
    μa ← randomSelection (U);    // Select a random user session
    randNumber ← genRandom (1, n) ;        // n is the number of
    requests in μa. This generates a random number between 1
    and n
    R ← getRequests (μa, randNumber); // Get a random number of
    user requests from the user session μa, and add them to
    the test suite
    foreach request ri ∈ R do
        Tnew ← Tnew ∪ {ri};
        newRequest ← searchRequestWithSameUrl (ri,R);
    end
    μb ← randomSelection (U) ;          // μb is different to μa
    if newRequest = ∅ then
        newRequest ← searchRequestWithSameUrl (ri, U);   // Search
        for a request rj in user session μa that has the same
        URL as the request ri from user session μa
    end
    if newRequest not found then
        Tnew ← Tnew ∪ {μb};     // If no such user session exists,
        then we use μb directly as a test case;
    else if newRequest is found in μb then
        Add to Tnew requests from newRequest onwards in μb;
    end
    Set μa as "used";
    U ← U \ {μa};
end
```

Algorithm 2. Test suite selection algorithm.

```
Data: The set of m user sessions, U = {μ1, μ2, ..., μm}
Result: The new test suite, Tnew
Tnew ← ∅;                                                    // Initialisation
while there are unused sessions in U do
    μa ← randomSelection (U);       // An unused session, μa, is
    selected from U;
    ri ← genRandomRequest (μa) ;     // An unused request, ri, is
    randomly selected from μa;
    if no unused requests left in μa then
        Tnew ← Tnew ∪ {μa} ;  // μa is directly as a test case if
        there are no more unused request left
    end
    if no name-value pair is present in ri then
        Label ri as "used";
    else
        rand ← genRandom (0,1) ;    // select randomly between the
        two options
        if rand > 0.5 then
            foreach NV in name-value pairs do
                NV ← deleteRandomChar (NV);
                Tnew ← Tnew ∪ {NV};
            end
        else
            newTest ← deleteRandomCharFromAllAtrings() ; // delete
            a character from all name-value pairs at once
            Tnew ← Tnew ∪ newTest ;      // form a single test case
        end
    end
    Set μa as "used";
    U ← U \ {μa};
end
```

10.1. Test case prioritisation and reduction for user session-based testing

Elbaum et al. propose two techniques for test suite size reduction. The first technique [36] is based on the test suite

reduction technique of Harold et al. [77], and is applied to the test cases which are generated from user sessions directly. The basic idea behind this technique is that some heuristics are applied to select the smallest subset t of the test suite T, which achieves the same functional coverage as T. Elbaum et al. show that for functional coverage, this technique reduces the test suite size by 98% while not detecting 3 out of 20 faults; for block coverage, it reduces test suite size by 93% while missing two faults; and for transition coverage, it reduces size by 79% while missing one fault. Thus, the overall effectiveness in terms of coverage and fault detection is reduced by a small margin ($\sim 2\%$ for functional coverage, $\sim 7\%$ for block coverage, and $\sim 20\%$ for transition coverage).

The second method used by Elbaum et al. [36] for reducing the test cases is based on clustering analysis. They group similar test cases in *clusters*, using a hierarchical agglomerative approach and Euclidian distance as the measure of similarity (similar to the procedure employed by Dickinson et al. [78]). They then generate a reduced test suite by randomly selecting one test case from each cluster. A smaller number of clusters provides greater test reduction at a cost of fault detection effectiveness. For example, when the cluster size is 4, the test suite is reduced by 98% and three faults are undetected. Clustering of user sessions is also discussed elsewhere [79], clustering is done according to a user session-based technique. In this case, the user sessions are clustered based on the service profile and then a set of representative user sessions are selected from a particular cluster. The user sessions are then further enhanced by adding additional requests, which take into account the dependence relationships between different Web pages. It was also found out, through two separate empirical analyses conducted, that the resulting test suite reduced the number of test cases, and also could detect a majority of the faults detected by the original test suite (greater than 90%) [79].

Sampath et al. [37] further extend on the test reduction techniques proposed by Elbaum et al. [36], by applying the *formal concept analysis* techniques [80,9]. Formal concept analysis aims at organising a set of objects O into a hierarchy. Besides the set of objects O, formal concept analysis takes as input a set of attributes A and a binary relation R known as a *context*, such that $R \subseteq O \times A$. The relationship R evaluates to boolean values and is true only if a set of objects, $o \subseteq O$ shares a set of attributes, $a \subseteq A$.

A tuple, t, for a subset of objects, O_i, and a subset of attributes, A_j, can be defined such that all and only the objects in O_i share all and only attributes in A_j. A *concept* is a tuple $t = (O_i, A_j)$ in which $O_i \subseteq O$ and $A_j \subseteq A$. In this technique, a concept lattice is first formed which is a *partial ordering* on the set of all concepts. The graph formed is a directed acyclic graph in which the nodes represent the concept and the edges denote the partial ordering. The top element $\top$ of the lattice represents the most general concept which represents the subset of attributes in A which are shared by all the objects O in the lattice. The bottom element $\perp$ represents the most special concept and represents the subset of objects in O that shares all the attributes in A.

In terms of user session-based testing, the set O represents the set of user sessions (i.e., the number of test cases) and the set A represents the URLs for the user sessions. Although an attribute comprises both URLs and name-value pairs for a user session μ_i, the effectiveness of choosing only URLs as the attribute set is demonstrated by Sampath et al. [37]. Thus, for a given pair of user session μ_i and URL r_j, the binary relation R is true iff μ_i requests r_j.

A lattice of user sessions is represented diagrammatically in Fig. 15. The different user sessions are shown in Table 1. The objects O are represented in rows, where the prefix "G", indicates GET requests, and the prefix "P" indicates POST requests), both of which prefix the different attributes A (columns of Table 1). The lattice shown in Fig. 15 is then constructed from the first 6 user sessions.

Once the lattice is formed, there are two techniques proposed by Sampath et al. to reduce the test suite, which are discussed in the following subsections.

10.2. Batch test suite reduction

This technique is similar to the clustering technique proposed by Elbaum et al. [36] because it selects the smallest set of user sessions (objects) similar to each other that exercises all the URLs which are executed by the original test suite. This technique also executes the common URL subsequences of different use cases as represented by the original test suite. The heuristic for selecting user sessions (or test cases) for this technique is to include a user session from each node next to the bottom element $\perp$, i.e., one level up from the bottom node $\perp$. These nodes are called *next-to-bottom* nodes. This is because the *similarity* of a set of user sessions U_i is defined by the number of attributes shared by all of the user sessions in U_i. Therefore, the set of objects which are closer to the bottom are *more highly similar* than the set of objects closer to the top of the lattice. If the bottom element $\perp$ is non-empty, the set of user sessions in $\perp$ are also included in the test suite. Thus, test cases are chosen from different clusters of similar use cases. The advantage of this technique is that the heuristic

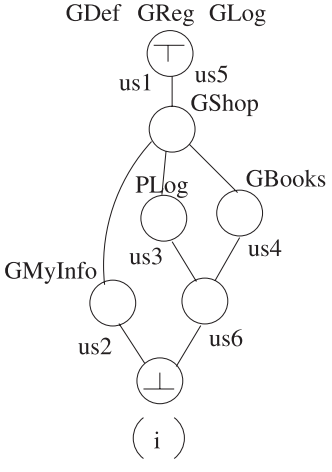

Fig. 15. A graphical representation of a lattice construction from the different objects and attributes given in Table 4. This figure is adapted from [37, Fig. 1].

Table 4
A tabular representation of the different objects and attributes used to construct a lattice. Only the first 6 user sessions are used for the construction of the lattice shown in Fig. 15. In this table, an "X" for a particular column/URL indicates that a user belonging to a specific user session accessed the URL. For example, the "X" symbols in the first row indicate that the user "us1" accessed/requested the URLs "GDef", "GReg", and "GLog". This table is adapted from [37, Fig. 1].

Session	GDef	GReg	GLog	PLog	GShop	GBooks	GMyInfo
us1	X	X	X				
us2	X	X	X		X		X
us3	X	X	X	X	X		
us4	X	X	X		X	X	
us5	X	X	X				
us6	X	X	X	X	X	X	
us7	X	X	X	X	X		X
us8	X		X			X	

is very simple to compute and label the *next-to-bottom* nodes, i.e., nodes that are included in the updated test suite. The limitation of the method is that all the user sessions are considered together, which may result in increased complexity of the overall approach.

10.3. Incremental reduced test suite update

This algorithm utilises the incremental concept formation algorithm developed by Godin et al. [80]. This technique starts with an initial concept lattice L and an initial reduced test suite T formed from an initial user session data set S. It then updates the test suite T' and the concept lattice L' as more user sessions are analysed. After every addition of a user session, the concept lattice L' is updated, and if a user session S_i replaces a *next-to-bottom* node in the lattice (i.e., the nodes which are connected to the bottom element $\perp$), S_i is then *added* to the updated test suite T' and the replaced *next-to-bottom* node representing user session is *removed* from the test suite T'. If, however, a user session S_i does not replace a *next-to-bottom* node in the lattice, the test suite is not changed and the user session (i.e., the test case) is ignored. The *existing* internal nodes, however, will never "sink" to the *next-to-bottom* nodes because the *partial ordering* of the existing internal nodes with respect to existing *next-to-bottom* nodes remain unchanged.

This is again demonstrated in Figs. 16 and 17, where the second lattice shows that the user session "us7" will be added to the test suite. However, in the third lattice, since "us8" does not replace a "next-to-bottom" node, it is not added to the test suite. The advantage of this approach is that not all the user sessions are considered together, which reduces the complexity of the overall approach.

Sampath et al. show that these techniques result in test suite size reduction by 87.8%, replay time reduction by 74.2%, 3.8% loss in statement coverage and no loss in function coverage. The effectiveness of the reduced test suite, however, was found to be 20% less than that of the original test suite [9].

Sampath et al. [37] also propose an automated prototype framework which enables the collection of user session data and hence helps in generation of reduced test suite. The test cases are replayed to generate coverage

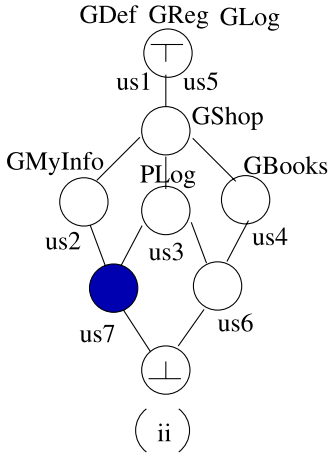

Fig. 16. A graphical representation of a new and updated lattice construction from two new user sessions, "us7" and "us8" (see Table 4). The new lattice includes user session "us7" as a *next-to-bottom-node*. The new user session "us7" will now be added to the updated test suite. This figure is adapted from [37, Fig. 1].

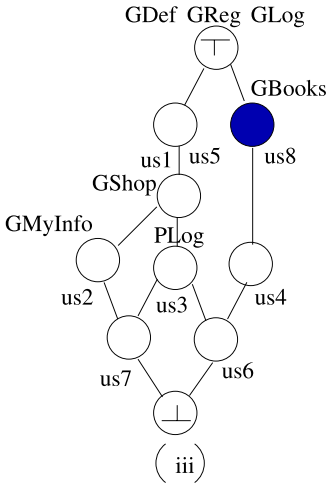

Fig. 17. The new lattice does not include user session "us8" (see Table 4) as a *next-to-bottom-node*. The new user session "us8" will therefore not be added to the updated test suite. This figure is adapted from [37, Fig. 1].

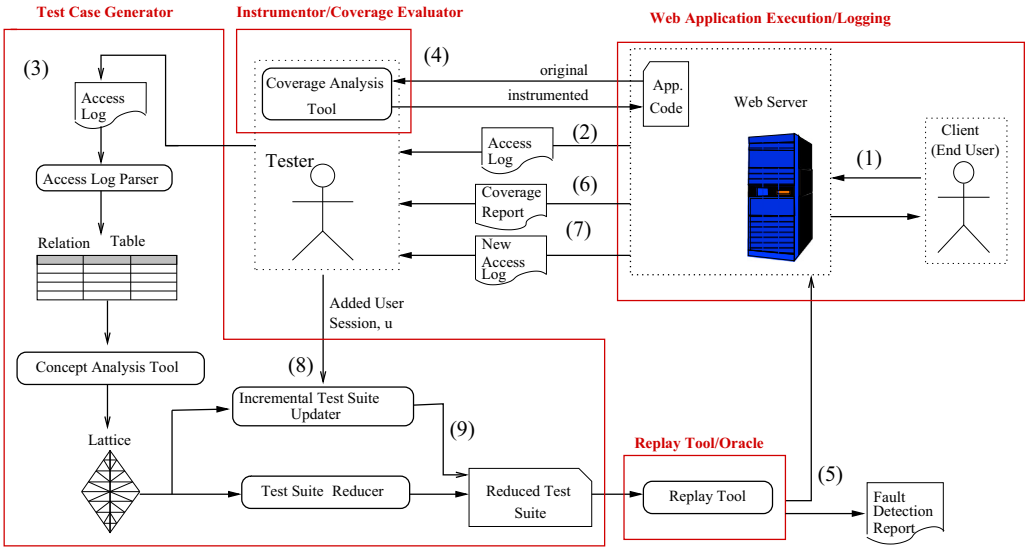

Fig. 18. Diagram depicting the framework proposed by Sampath et al. [37]. This figure is taken from [37, Fig. 3].

reports and perform fault detection analysis. As shown in Fig. 18, the process begins with the collection of user session data, in Step 1 in the diagram. This phase is known as the *Web application execution or logging* phase. Sampath et al. [37] develop a modified version of the Resin[21] Web server's *AccessLog* class to collect user session data attributes of interest. These include information relating to each user request, such as IP address, timestamp, requested URL (s), GET/POST data and cookies. Timestamp of a given request r_i is tracked. A user session begins when a request arrives from a new IP address and ends when the user leaves the Web site or the session times out. If requests are more than 45 min apart, then they are considered to be requests from distinct sessions. The access log data is then provided to a tester (Step 2 in the diagram). Then the *access log* is parsed to generate a relation table, similar to the one depicted in Fig. 15. The test suite reducer implements the heuristic (Batch test suite reduction) for obtaining a reduced test suite. The concept analysis tool depicted in the figure outputs a lattice.

The *test coverage evaluator* (Step 4) comprises the Java code coverage tool "Clover",[22] which instruments the Java files that are generated by the server. The reduced test suite is then replayed (Step 5). The GNU Unix utility "wget" was used as the replay tool. Input parameters to the "wget" utility include cookie information and POST/GET data associated with the request to maintain the application state information. To guarantee consistency in replay, the database is also restored to its original state before replay. The coverage report is then generated (Step 6). The test suite is then incrementally updated in Steps 7, 8 and 9 with new user sessions coming from *New Access Log* (Step 7) and the current lattice L getting updated and the current test suite T getting updated (Step 9).

10.4. Test case reduction through examining URL trace

Another technique for reduced test case selection is proposed by Zhongsheng [38]. It involves identifying whether a URL trace (or a sequence of URL requests) in a particular user session U_α is the *prefix* of a URL trace requested by another user session U_β. A trace α is the prefix of another trace β iff α is the subsequence of β and they have the same initial symbol (or request). For example, if $\alpha = abc$, $\beta = abcde$, then α is a prefix of β). If a URL trace in one user session U_α is the prefix of another URL trace in U_β, then the user session U_α is removed from the test case. The number of user sessions required for testing from this algorithm can be reduced greatly. The algorithm can also be easily implemented. It also covers all the URLs requested by the original set of user sessions and keeps the sequence of URL requests, i.e., it guarantees that the original test requirements are satisfied.

The user sessions reduced by the algorithm are then further divided into subgroups, each of which is regarded as an individual test suite. In order to group these user sessions together, different discontinuous threshold values, to compare against the lengths of greatest common prefixes of user sessions, and denoted by $\zeta_1, \zeta_2, \ldots, \zeta_k$, such that $\zeta_i \geq 1$ for $1 \leq i \leq k$, are defined. All user sessions whose *greatest common prefix* (the longest common prefix between different user sessions) lies between certain threshold values are grouped together. For example, let us consider three different threshold values, $\zeta_1, \zeta_2, \zeta_3$, such that $\zeta_1 = 2$, $\zeta_2 = 4$, and $\zeta_1 = 7$. Let us also consider four different user traces, such that $\gamma_1 =$ "abcdefg", $\gamma_2 =$ "abcdeh", $\gamma_3 =$ "abcd", and $\gamma_4 =$ "cde". Let us denote the length of the greatest common prefix of a particular group of user traces by α. Then, depending on the threshold values, $\zeta_1, \zeta_2, \zeta_3$, we

[21] http://www.caucho.com/resin/. This site was last accessed on January 31, 2013.

[22] http://www.thecortex.net/clover/. This site was last accessed on January 31, 2013.

can divide these traces into two groups/sets, such that the values of α for these groups lie within the range of the threshold values, $\zeta_1, \zeta_2, \zeta_3$. One way to split these user traces could be to group it into two sets, namely, $\{\gamma_1, \gamma_2\}$, where the length of the greatest common prefix (the string "abcde") is 5, which is between ζ_2 and ζ_3 (or more formally, $4 < |\alpha| \leq 7$), and $\{\gamma_3, \gamma_4\}$, where the length of the greatest common prefix is 0, which is less than the threshold value of ζ_1 (or, more formally, $|\alpha| \leq 2$). The user traces could also have been split differently into two groups, namely, $\{\gamma_1, \gamma_2, \gamma_3\}$, where the length of the greatest common prefix (the string "abcd") is 4, which is between ζ_1 and ζ_2 (or more formally, $2 < |\alpha| \leq 4$), and $\{\gamma_4\}$, where the length of the greatest common prefix is 0, which is less than ζ_1 (or, more formally, $|\alpha| \leq 2$) [38].

The test suite with the *shortest* greatest common prefix is executed first [38]. This is because these user sessions indicate special or different URL requests, which represent distinct requirements for Web applications. Since these requests represent the sequence of URLs which is rarely executed in user interactions, it is especially important to test these scenarios as something could easily go wrong. The rest of the test suites are prioritised according to their descending lengths of their greatest common prefixes, i.e., the test suite with the longest greatest common prefix is executed, then the one with second longest greatest common prefix, and so on. In each test suite, the test cases are prioritised according to the coverage ratios of URLs requested, i.e., the test case with longer URL trace requested is executed earlier. If the lengths of URL traces of several test cases in the same test suite are equal, the order of execution is randomly determined.

11. Conclusion and future directions

The World Wide Web has become an indispensable part of our society in two short decades. With the rapid advances in hardware infrastructure and software technologies, Web applications, and software that runs on a Web server enabling users to interact with the application via a browser or other software services have grown to be so sophisticated that it supports complex interactions with users (and other Web applications). Additionally, these applications are also able to complete complex transactions in a secure manner in a relatively short period of time. The ubiquity of the Web and the central role Web applications on the Web play make it imperative to ensure that Web applications are secure, correct and efficient.

Testing is a validation and quality assurance approach widely practised by companies. Software testing in general and Web application testing specifically have also been an active research area. Despite the importance of Web applications, major research efforts in the testing of Web applications have not been surveyed in a substantial way. In this paper, we present some recent advances in Web testing techniques and discuss the strengths and weaknesses of these techniques.

Of the different techniques, some are more effective at finding faults, including scanning and crawling techniques, mutation testing and fuzz testing, in existing applications, whereas others are more effective at ensuring that the application has been adequately tested. In other words, different testing techniques have different goals and targets, and thus some testing techniques may be more adequate than others depending on the nature of the testing that needs to be performed on a Web application. Moreover, each of these techniques differs in their inputs, outputs, conditions for stopping the test, and their primary purpose (as summarised in Tables 1–3 in Section 2). Additionally, we also discuss how the different techniques can be applied to Web applications and ensure that the Web applications have been adequately tested.

As we have already discussed, due to the heterogeneous nature and the ever growing complexity of Web applications, it is important that proper attention is given to testing Web applications. Also, since a large number of users use Web applications to carry out a host of different tasks, including financial transactions, appointment booking and communications (emails, instant messaging and voice/video calls), it is important to ensure the privacy of users and the integrity of the transactions. Therefore, if a Web application is not properly tested, it means that a lot of potential users could risk losing their private data and/or facing severe financial losses. One of the most critical areas of Web application testing – and one that can cause severe financial and data losses if compromised or not done rigorously enough – is Web security. As we described in Section 6, there are several scanning and crawling techniques which test specifically to test if a given Web application contains security flaws by injecting unsanitised inputs. Some of these scanners are also open-source and readily available, including SecuBat [8], Selenium [61], and JsUnit, which we discussed in the paper. In addition, we also discussed Mutation testing in Section 4, where lines of code in a piece of software are modified and then test cases are deemed successes or failures depending on whether or not they can detect the change in the software code. All these techniques expose that the security flaws in Web applications, thus ensuring safer and more robust Web applications.

Moreover, we have also discussed various testing techniques which ensure that the application functions consistently with the specification(s) as required. Some of these techniques include creating an overall model of the application, from which test cases are derived for the test suite, such as Graph based techniques [19], the Finite State Machine [20], and the probable FSM [21]. Other techniques to ensure that the application behaves consistently with the specification include search-based software engineering techniques [23], which ensure that a Web application is tested as thoroughly as possible, as measured by the coverage of its branches; Concolic testing techniques [73] – such as DART [65], EXE [69] – combine concrete execution and symbolic execution to ensure that the program traverses along different paths/branches. Furthermore, we have also discussed how concolic testing can be applied to test PHP Web Applications [34]. In addition, we also discussed user session-based testing techniques in Section 10, in which test cases are derived from data collected from user sessions (URLs and name-value pairs).

However, there is also significant future work and research that can be done on testing of Web applications.

In the case of user session-based testing, the potential benefits of integrating user session data with traditional testing techniques have yet to be fully realised and requires further investigation [36]. Furthermore, the technique could also be extended to take into account concurrent user requests and keep track of Web application states. Similarly, in order to truly exploit the power of user-session data, user-session data could be used to assess the appropriateness of an existing test suite in the face of shifting operational profiles [36]. In addition, a more holistic and empirical analysis of techniques proposed to reduce the number of user sessions in a test suite could be done in future (e.g., by conducting experiments on a larger set of URLs, etc.) [37].

Similarly, in the case of the finite state machine-based testing technique [20], further work needs to be done with regards to automating the testing technique. Another limitation of the Finite State Machine-based (FSM-based) testing technique is that it has limited support for unanticipated, user-controlled transitions, called operational transitions [81]. This includes a user going directly to an internal Web page with a bookmark, URL rewriting and unanticipated back and forward navigation. In order to test these transitions, a method will have to be found to model operational transitions in a tractable manner, that is, there are no space explosion problems. This could be as simple as keeping a list of potential operational transitions and selecting them at various points in a test sequence [20]. This issue has already been discussed to some extent in Wu and Offutt [81], and therefore future work could involve combining these two techniques.

In the case of scanning and crawling techniques, more plug-ins to initiate different types of attacks could be added to a scanner [8]. Furthermore, most of the scanning techniques discussed in this paper (e.g., SecuBat, WAVES) do not consider the number or lengths of queries required to detect a flaw in a Web application. More work could be done in this area, perhaps in combination with similar works done by Madhavan et al. [82], to reduce the length of query strings when accessing the deep Web [82] (i.e., the part of the Web that is hidden from the user and acts as the back-end for most Web applications, e.g., the database).

References

[1] G.J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, Wiley, New Jersey, USA, 2011.

[2] T. Berners-Lee, The world-wide web, Comput. Netw. ISDN Syst. 25 (1992) 454–459.

[3] C.S. Wallace, Digital computers, in: S.T. Butler, H. Messel (Eds.), Atoms to Andromeda, Shakespeare-Head, Sydney, 1966, pp. 215–245.

[4] G.A. Di Lucca, A.R. Fasolino, Testing web-based applications: the state of the art and future trends, Inf. Softw. Technol. 48 (2006) 1172–1186.

[5] M.H. Alalfi, J.R. Cordy, T.R. Dean, Modelling methods for web application verification and testing: state of the art, Softw. Test. Verif. Reliab. 19 (2009) 265–296.

[6] D. Flanagan, JavaScript: The Definitive Guide, O'reilly, California, USA, 2011.

[7] R.J. Lerdorf, K. Tatroe, B. Kaehms, R. McGredy, Programming Php, 1st ed. O'Reilly & Associates, Inc, Sebastopol, CA, USA, 2002.

[8] S. Kals, E. Kirda, C. Kruegel, N. Jovanovic, SecuBat: a web vulnerability scanner, in: Proceedings of the 15th International Conference on World Wide Web, WWW '06, ACM, New York, NY, USA, 2006, pp. 247–256.

[9] S. Sampath, V. Mihaylov, A. Souter, L. Pollock, A scalable approach to user-session based testing of Web applications through concept analysis, in: Proceedings of the 19th International Conference on Automated Software Engineering, 2004, pp. 132–141.

[10] S. Elbaum, G. Rothermel, S. Karre, M. Fisher II, Leveraging user-session data to support Web application testing, IEEE Trans. Softw. Eng. 31 (2005) 187–202.

[11] T. O'Reilly, What is web 2.0: design patterns and business models for the next generation of software, Commun. Strateg. (2007) 17.

[12] J.J. Garrett, et al., Ajax: a new approach to web applications, 2005. ⟨http://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php⟩.

[13] A. Mesbah, A. van Deursen, Invariant-based automatic testing of AJAX user interfaces, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 210–220.

[14] W.J. Chun, Core Python Programming, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[15] D. Flanagan, Y. Matsumoto, The Ruby Programming Language, first ed. O'Reilly, California, USA, 2008.

[16] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, D. Song, A symbolic execution framework for JavaScript, in: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP'10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 513–528.

[17] C.S. Wallace, Statistical and Inductive Inference by Minimum Message Length, Springer-Verlag, New York, USA, 2005.

[18] D.J. Mayhew, The usability engineering lifecycle, in: CHI 98 Conference Summary on Human Factors in Computing Systems, CHI '98, ACM, New York, NY, USA, 1998, pp. 127–128.

[19] F. Ricca, P. Tonella, Analysis and testing of Web applications, in: Proceedings of the 23rd International Conference on Software Engineering, ICSE '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 25–34.

[20] A.A. Andrews, J. Offutt, R.T. Alexander, Testing Web applications by modeling with FSMs, Softw. Syst. Model. 4 (2005) 326–345, http://dx.doi.org/10.1007/s10270-004-0077-7.

[21] Z. Qian, H. Miao, Towards testing web applications: a PFSM-based approach, in: Advanced Materials Research, 2011, vol. 1, pp. 220–224.

[22] U. Praphamontripong, J. Offutt, Applying mutation testing to web applications, in: ICST Workshops, 2010, pp. 132–141.

[23] N. Alshahwan, M. Harman, Automated web application testing using search based software engineering, in: Automated Software Engineering, ASE, 2011, pp. 3–12.

[24] Y.-W. Huang, S.-K. Huang, T.-P. Lin, C.-H. Tsai, Web application security assessment by fault injection and behavior monitoring, in: Proceedings of the 12th International Conference on World Wide Web, WWW '03, ACM, New York, NY, USA, 2003, pp. 148–159.

[25] J. Bau, E. Bursztein, D. Gupta, J. Mitchell, State of the art: automated black-box web application vulnerability testing, in: IEEE Symposium on Security and Privacy, 2010, pp. 332–345.

[26] M. Benedikt, J. Freire, P. Godefroid, Veriweb: automatically testing dynamic web sites, in: Proceedings of 11th International World Wide Web Conference (WWW 2002), 2002.

[27] L. Zhou, J. Ping, H. Xiao, Z. Wang, G. Pu, Z. Ding, Automatically testing web services choreography with assertions, in: Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering, ICFEM'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 138–154.

[28] L. Frantzen, M. Las Nieves Huerta, Z.G. Kiss, T. Wallet, Web Services and Formal Methods, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 143–157.

[29] S. Artzi, J. Dolby, S.H. Jensen, A. Møller, F. Tip, A framework for automated testing of JavaScript web applications, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 571–580.

[30] P. Heidegger, P. Thiemann, Contract-driven testing of JavaScript code, in: Proceedings of the 48th International Conference on Objects, Models, Components, Patterns, TOOLS'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 154–172.

[31] P. Godefroid, M. Levin, D. Molnar, Automated whitebox fuzz testing, in: Network & Distributed System Security Symposium, NDSS, 2008.

[32] P. Saxena, S. Hanna, P. Poosankam, D. Song, FLAX: systematic discovery of client-side validation vulnerabilities in rich web applications, in: 17th Annual Network & Distributed System Security Symposium (NDSS), 2010.

[33] P. Godefroid, A. Kiezun, M.Y. Levin, Grammar-based whitebox fuzzing, SIGPLAN Not. 43 (2008) 206–215.

[34] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, M.D. Ernst, Finding bugs in dynamic web applications, in: Proceedings of the

2008 International Symposium on Software Testing and Analysis, ISSTA '08, ACM, New York, NY, USA, 2008, pp. 261–272.
[35] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, Z. Su, Dynamic test input generation for web applications, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, ACM, New York, NY, USA, 2008, pp. 249–260.
[36] S. Elbaum, S. Karre, G. Rothermel, Improving web application testing with user session data, in: Proceedings of the 25th International Conference on Software Engineering, ICSE '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 49–59.
[37] S. Sampath, A. Souter, L. Pollock, Towards defining and exploiting similarities in Web application use cases through user session analysis, IEE Semin. Dig. 2004 (2004) 17–24.
[38] Z. Qian, Test case generation and optimization for user session-based web application testing, J. Comput. 5 (2010).
[39] R. Binder, Testing Object-oriented Systems: Models, Patterns, and Tools, Addison-Wesley, Reading, MA, 1999.
[40] J. Offutt, S. Liu, A. Abdurazik, P. Ammann, Generating test data from state-based specifications, Softw. Test. Verif. Reliab. 13 (2003) 25–53.
[41] P. Chevalley, P. Thenod-Fosse, A mutation analysis tool for Java programs, Int. J. Softw. Tools Technol. Transf. 5 (2003) 90–103, http://dx.doi.org/10.1007/s10009-002-0099-9.
[42] Y.-S. Ma, Y.-R. Kwon, J. Offutt, Inter-class mutation operators for java, in: Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 352–.
[43] P.E. Black, V. Okun, Y. Yesha, Mutation operators for specifications, in: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ASE '00, IEEE Computer Society, Washington, DC, USA, 2000, pp. 81–88.
[44] J. Offutt, Y.-S. Ma, Y.-R. Kwon, The class-level mutants of MuJava, in: Proceedings of the 2006 International Workshop on Automation of Software Test, AST '06, ACM, New York, NY, USA, 2006, pp. 78–84.
[45] J. Offutt, Y. Wu, Modeling presentation layers of web applications for testing, Softw. Syst. Model. 9 (2010) 257–280, http://dx.doi.org/10.1007/s10270-009-0125-4.
[46] M. Harman, 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 '07, ACM, New York, NY, USA, 2007, pp. 73–83.
[47] B. Korel, Automated software test data generation, IEEE Trans. Softw. Eng. 16 (1990) 870–879.
[48] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun, J. Wegener, Input domain reduction through irrelevant variable removal and its effect on local, and hybrid search-based structural test data generation, IEEE Trans. Softw. Eng. 38 (2012) 453–477.
[49] N. Tracey, J. Clark, K. Mander, J. McDermid, An automated framework for structural test-data generation, in: Proceedings of the 13th IEEE International Conference on Automated Software Engineering, ASE '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 285–288.
[50] G. Navarro, A guided tour to approximate string matching, ACM Comput. Surv. 33 (1999) 2001.
[51] M. Alshraideh, L. Bottaci, Search-based software test data generation for string data using program-specific search operators: research articles, Softw. Test. Verif. Reliab. 16 (2006) 175–203.
[52] L.K. Shar, H.B.K. Tan, Defending against cross-site scripting attacks, IEEE Comput. 45 (2012) 55–62.
[53] J. Cho, H. Garcia-Molina, Parallel crawlers, in: Proceedings of the 11th International Conference on World Wide Web, WWW '02, ACM, New York, NY, USA, 2002, pp. 124–135.
[54] R.C. Miller, K. Bharat, SPHINX: a framework for creating personal, site-specific Web crawlers, Comput. Netw. ISDN Syst. 30 (1998) 119–130.
[55] C. Bowman, P.B. Danzig, D.R. Hardy, U. Manber, M.F. Schwartz, The harvest information discovery and access system, Comput. Netw. ISDN Syst. 28 (1995) 119–125.
[56] U. Manber, M. Smith, B. Gopal, WebGlimpse: combining browsing and searching, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC 97, USENIX Association, Berkeley, CA, USA, 1997, pp. 15–15.
[57] E.V. Nova, D. Lindsay, Our Favorite XSS Filters and How to Attack Them, 2009. ⟨http://www.blackhat.com/presentations/bh-usa-09/VELANAVA/BHUSA09-VelaNava-FavoriteXSS-SLIDES.pdf⟩.
[58] K. Moody, M. Palomino, SharpSpider: spidering the web through web services, in: Proceedings of the First Conference on Latin American Web Congress, LA-WEB '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 219–221.
[59] D. Richardson, M. Thompson, The RELAY model of error detection and its application, in: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, IEEE, Banff, Canada, 1988, pp. 223–230.
[60] L. Morell, Theoretical insights into fault-based testing, in: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, 1988, pp. 45–62.
[61] G. Gheorghiu, A look at Selenium, Softw. Qual. Eng. 7 (2005) 38–44.
[62] A. Mesbah, E. Bozdag, A. v. Deursen, Crawling AJAX by inferring user interface state changes, in: Proceedings of the 2008 Eighth International Conference on Web Engineering, ICWE '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 122–134.
[63] A. Mesbah, A. van Deursen, S. Lenselink, Crawling Ajax-based web applications through dynamic analysis of user interface state changes, ACM Trans. Web 6 (2012) 3:1–3:30.
[64] B.P. Miller, G. Cooksey, F. Moore, An empirical study of the robustness of MacOS applications using random testing, in: Proceedings of the 1st International Workshop on Random Testing, RT '06, ACM, New York, NY, USA, 2006, pp. 46–54.
[65] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, SIGPLAN Not. 40 (2005) 213–223.
[66] A.J. Offutt, J.H. Hayes, A semantic model of program faults, SIGSOFT Softw. Eng. Notes 21 (1996) 195–200.
[67] L. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: C. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 4963, Springer, Berlin, Heidelberg, 2008, pp. 337–340.
[68] A. Biere, A. Cimatti, E. Clarke, O. Strichman, Y. Zhu, Bounded model checking, Adv. Comput. 58 (2003).
[69] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, D.R. Engler, EXE: automatically generating inputs of death, ACM Trans. Inf. Syst. Secur. 12 (2008) 10:1–10:38.
[70] E. Schwartz, T. Avgerinos, D. Brumley, All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), in: IEEE Symposium on Security and Privacy (SP), IEEE, Washington, DC, USA, 2010, pp. 317–331.
[71] J. Newsome, D.X. Song, Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, in: NDSS, The Internet Society, 2005.
[72] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, P. Barham, Vigilante: end-to-end containment of internet worms, in: ACM SIGOPS Operating Systems Review, 2005, vol. 39, ACM, New York, NY, USA, pp. 133–147.
[73] K. Sen, Concolic testing, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, ACM, New York, NY, USA, 2007, pp. 571–572.
[74] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, in: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, ACM, New York, NY, USA, 2005, pp. 263–272.
[75] V. Ganesh, D. Dill, A decision procedure for bit-vectors and arrays, in: W. Damm, H. Hermanns (Eds.), Computer Aided Verification, Lecture Notes in Computer Science, vol. 4590, Springer, Berlin, Heidelberg, 2007, pp. 519–531.
[76] A. Futoransky, E. Gutesman, A. Waissbein, A dynamic technique for enhancing the security and privacy of web applications, in: Proceedings of Black Hat USA, 2007.
[77] M. Harold, R. Gupta, M. Soffa, A methodology for controlling the size of a test suite, Trans. Softw. Eng. Methodol. 2 (1993) 270–285.
[78] W. Dickinson, D. Leon, A. Podgurski, Pursuing failure: the distribution of program failures in a profile space, in: Proceedings of the 8th European Software Engineering Conference, ACM Press, New York, NY, USA, 2001, pp. 246–255.
[79] X. Luo, F. Ping, M.-H. Chen, Clustering and tailoring user session data for testing web applications, in: International Conference on Software Testing Verification and Validation, ICST '09, 2009, pp. 336 –345.
[80] R. Godin, R. Missaoui, H. Alaoui, Incremental concept formation algorithms based on Galois concept lattices, Comput. Intell. 11 (1995) 246–267.
[81] Y. Wu, J. Offutt, X. Du, Modeling and testing of dynamic aspects of Web applications, Technical Report ISE-TR-04-01, Department of Information and Software Engineering, George Mason University, 2004.
[82] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, A. Halevy, Google's Deep Web crawl, in: Proceedings of the VLDB Endowment, vol. 1, 2008, pp. 1241–1252.