2011 35th IEEE Annual Computer Software and Applications Conference Workshops

Automated Test Case Generation for Web Applications from a Domain Specific Model

Arne-Michael Törsel
Fachhochschule Stralsund
Zur Schwedenschanze 15
18435, Stralsund, Germany
Email: Arne-Michael.Toersel@fh-stralsund.de

***Abstract*—Model-based testing is a promising technique for test case design that is used in an increasing number of application domains. However, to fully gain efficiency advantages, intuitive domain-specific notations with comfortable tool support as well as a high degree of automation in the whole testing process are required. In this paper, a model-based testing approach for web application black box testing is presented. A notation for web application control flow models augmented with data flow information is introduced. The described research prototype demonstrates the fully automated generation of ready to use test case scripts for common test automation tools including test oracles from the model.**

***Index Terms*—test automation, web applications, model-based testing, automated test oracles**

I. Introduction

There have been significant advances in test automation for black box testing of web applications using commercial or open source test automation tools like *Selenium*[1] or *Canoo Webtest*[2]. These tools execute the test cases automatically, steered by script implementations that describe simulated user interaction and verification steps. A major problem that remains unresolved is how to create and maintain these test cases in an efficient way, as at present this is generally done manually by the test engineer. Modularization and abstraction of test script code for reuse purposes partially alleviates this problem, but maintenance is still a problem with large automated test suites. Kaner et al. [1], for example, identified high test script maintenance costs as the major reason for failing test automation efforts on the user interface level.

Recently, the usage of models to derive test cases, designated as *model-based testing*, has also been proposed for the domain of web application black box testing [2]–[5]. The idea of model-based testing is to create and maintain a model that contains information about the structure and possibly also about the desired behaviour of the application. Test cases are then derived either manually or automatically from the model with algorithms that systematically cover the model using so called *selection criteria*. Instead of having to maintain the test suite, one updates the model and regenerates the test cases when necessary. Clearly, these test cases can only test aspects of the application that have been modelled. Especially the problem of automatically creating a test oracle, a mechanism to determine the test outcome, requires that the necessary behavioural information is encoded in the model. If that information is not available in the model, test approaches require test oracles to be added manually to the generated test cases or only weak test oracles, for example, checks for detectable crashes of the tested system, can be created automatically. It must also be noted that due to the need for abstraction, one usually can not cover all test requirements with one model type. Different model types for different test purposes are needed or the model-based testing process needs to be complemented by other test approaches. Another consequence of abstraction is that the generated test cases are on the model level of abstraction and must be adapted and augmented to be applicable to the actual system under test.

Comparing the efficiency of model based testing and 'traditional' scripted test automation requires considering various cost and effort aspects of the particular approach. Firstly, the effort to create and maintain the model itself needs to be considered. The high level of abstraction generally means that modelling is intellectually more demanding than implementing test scripts and requires a larger initial investment in tester education. Good modelling tool support and an intuitive modelling notation can support the test engineer in this case. Empirical research indicates [6], that one must provide tools that match at least the comfort level the prospective user is used to, if a new modelling notation is to be introduced to ensure user acceptance. Secondly, an effort is required to derive test cases from the model. The adaptation of these derived abstract test cases to the system under test is also a cost factor that needs to be considered. Manual work for test case derivation and adaptation might not be feasible in scenarios where the model is subject to frequent change as in agile development processes. Therefore, automated methods for test case selection and adaptation should be employed. Finally, the level of tool support for tracing back errors found during test execution to the model and ultimately to the relevant application parts (this is referred to as *traceability*) may have a significant impact on the total costs of the model based testing approach [7]. In summary, for model based testing to be more efficient than traditional script based test automation approaches a large amount of automation in the whole process from the model to the executable test case and back as well

[1]www.seleniumhq.org
[2]webtest.canoo.com

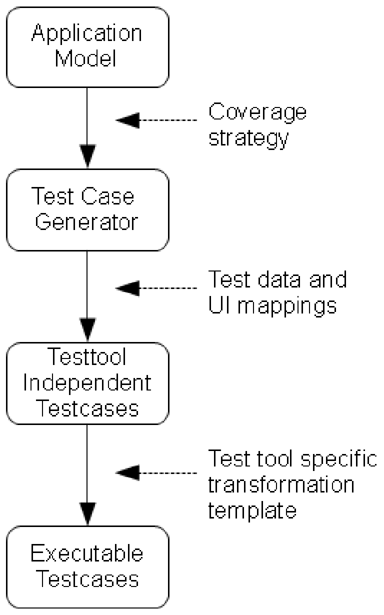
Application Model
Coverage strategy
Test Case Generator
Test data and UI mappings
Testtool Independent Testcases
Test tool specific transformation template
Executable Testcases

Fig. 1. Schema of proposed testing approach

as solid tool support are necessary.

Given these prerequisites, in this paper a model based black box testing approach for web applications that demonstrates a high degree of automation along the whole process from the model to executable test cases for test automation tools is proposed. It builds upon ideas from previous work on model-based testing of web applications [2]–[4] with the addition of information to the meta-model that allows web page content related test oracles to be automatically deduced and transformed to an executable format during test case generation. The generated abstract, tool independent test cases are automatically transformed to tool specific test scripts for execution and evaluation. Due to this high degree of automation the proposed approach may help to reduce the maintenance effort for web application test suites. Figure 1 gives a schematic overview of the workflow. The presented approach was implemented in a prototype using the Eclipse Modelling project technologies *Xtext* and *Xpand* to offer comfortable tool support especially for editing the model.

II. Model Design

Considering the target to generate executable test cases for test automation tools from a model with a high degree of automation within the entire process, four major requirements were identified:

1) The model must reflect the application structure, that is web pages and navigational connections between them, to allow a generator algorithm to systematically traverse it to create test sequences and to measure and reach structural coverage
2) The model must contain information about the dynamic behaviour of the application, that is the flow of data and its manipulation as well as data related dependencies between workflows, to enable the generation of user interface observable test oracles and to be able to calculate an applicable test case ordering
3) A method to supplement test input data and to connect test data pools to model workflows
4) A support mechanism to enable automatic bridging of the gap between the generated test tool independent test cases and test scripts that can be executed using a test tool directly on the system under test

These requirements are met by the proposed meta-model. The first requirement is fulfilled by logically modelling the application as a directed graph. This reflects the fundamental user request / server response control flow scheme of typical web applications. A server response yields a *view* of the application and is represented by a vertex in the graph. For every possible *transition* from one view to another view there is a directed edge in the graph. In a web application such a transition is a request usually caused either by the user clicking a link or submitting a form. Loops in the graph, i. e. transitions with identical start and target vertices, are possible and represent workflows where the user remains on the same view but the internal state of the application may have changed as a result of the interaction.

To satisfy the second requirement, the model can also contain variables of the basic types String, Number and Boolean to model the application state. Variables can be used in expressions using basic mathematical and boolean operators. Transitions can carry out assignments of simple values or expressions to variables. Transitions can also be guarded against usage during test case generation by conditional expressions. In this way, assignments and guards on transitions are used to model application state related dependencies between paths in the graph. In addition, variable values are used during test case generation to formulate test oracles (see Section *Test Case Generation* for more details). Variables are declared in two distinct scopes, either *Permanent* or *Session*. Whenever the web application session is reset during test case generation, the session-scoped variables revert to their initial values while permanent-scoped variables keep their current value. Permanent variables therefore represent the modelled persistent state of the application that is independent of the current application session.

The third requirement is fulfilled by introducing the *Tuple* variable type which allows external test input data to be associated with the model. A tuple variable can be seen as a read-only set of key-value pairs, which can be accessed likewise the primitive variable types and used in expressions. This also allows to express input data related guard conditions. The tuple variable fields can be set using external test data providers. Various provider implementations are possible through a plugin mechanism. The realization of the fourth requirement, support for automatic adaptation of abstract test cases to the abstraction level of the application and to specific test automation tools, is described in detail in Section *Test Case Transformation.*

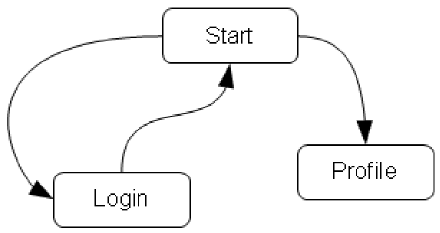

Fig. 2. Example view transition graph

Listing 1. Model example in the prototype
```
 1  Permanent-State:
 2  Tuple user = "UserDataSource"

 4  Session-State:
 5  Boolean loggedIn = "false"

 7  Views:
 8  View Start {
 9    Text "Welcome back!" When "loggedIn == true"
10    Text "Please log in" When "loggedIn == false"

12    Transition Login {
13      When "loggedIn == false"
14      Link "login"
15    }

17    Transition Profile {
18      When "loggedIn == true"
19      Link "profile"
20    }
21  }

23  View Login {
24  Text "Please enter login information"

26    Transition Start {
27      Input "username" = "{user.username}"
28      Input "password" = "{user.password}"
29      Button "login.submit"
30      Assign loggedIn = "true"
31    }
32  }

34  View Profile {
35    Text "Profile for {user.username}"
36    ...
37  }
```

III. Model Example

The primary modelling notation in the prototype is textual. Using the Eclipse Xtext framework, an editor was generated from the meta-model that enables comfortable model editing in the common Eclipse environment. Furthermore, as Xtext is based on the Eclipse Modeling Framework (EMF), a graphical editor could be implemented relatively easy as well using the Eclipse Graphical Modelling Project (GMP). The example in Listing 1 demonstrates the basic elements of the meta-model. A simple application scenario consisting of three views - *Start*, *Login* and *Profile* - is modelled (see a graph of the view transition structure in Figure 2). To access the *Profile* view, the user has to log in by supplying a username and a password at the view *Login* before.

The model starts with the two sections for variable declaration *Permanent-State* at line 1 and *Session-State* at line 4. Using the keyword *Tuple*, a tuple variable named *user* is declared at line 2. It is bound to an external data source named UserDataSource, which could be a spreadsheet table for example. Another variable of type *Boolean* named *loggedIn* is declared at line 5 which is session scoped and holds the login state of the user. The main part of the model consists of the declaration of three views in the *Views* section starting at line 7: the view *Start* at line 8, *Login* at line 23 and *Profile* at line 34.

From view *Start* there are two possible transitions declared by the keyword *Transition* to the views *Login* (line 12) and *Profile* (line 17). Both transitions contain a guard condition declared by the keyword *When* at lines 13 and 18. More conditions could be added using the keyword *When* several times. The expressions of these guard conditions state, that the transition to view *Login* can only be used if the value of variable *loggedIn* is false and to view *Profile* if *loggedIn* is true. The two transitions are triggered by the user clicking a link on the *View* which is expressed by using the keyword *Link*. The parameter of the *Link* keyword is a symbolic name which refers to a real HTML link in the tested application (see Section *Test Case Transformation*). Initially, only the transition to the view *Login* is usable because at line 5 the variable *loggedIn* is initialized to the value false.

At the view *Login* there is only one transition back to the view *Start*. This transition models a login form, where the user must input a username and a password. The necessary input is modelled using the keyword *Input*. The first parameter denotes a symbolic name for the input field. The expression after the equation sign is the actual value to be used as input. In that case values from the *Tuple* variable *user* are used by referencing the variable field in curly braces. The transition is triggered by a button which is modelled by the keyword *Button*. Similar to links the parameter is a symbolic name for the button. At line 30 as a result of the login procedure the value of the variable *loggedIn* is set to true after the transition has been used to get back to view *Start* by using the keyword *Assign*. The assignment causes the transition at line 12 to become unusable and the transition at line 17 to become usable due to the transition guards checking the variable value.

Finally, the *Text* keyword is used to declare text expressions that should be present in the HTML source code of the containing view. This is a means of deriving test oracles (see Sections *Test Case Generation* and *Test Case Transformation* for details). At line 9 and 10 there are two *Text* directives that are evaluated only conditionally, depending on the result of the check performed by the following *When* expression. In this way, different expressions can be searched for, depending on application state that is modelled using variables. The two *Text* expressions for the view *Start* are static. In contrast the *Text* expression at the view *Profile* declared at line 35 is dynamic, as the field username of *Tuple* variable user is inserted with the expression in curly braces. When traversing the model for test case generation, the current variable value is always used to resolve the expression.

IV. Test Case Generation

This section gives a short overview of the test case generation algorithm. A subsequent publication will contain a more detailed description. Initially, the model parser constructs a directed graph structure of the model views (vertices) and transitions (edges), the *application graph*. An extended breadth-first search (BFS) is used to explore that graph from the designated *start view*. The start view is typically the 'home page' of the web application. In every vertex all paths leading to it from the start view using either only the BFS spanning tree edges or paths ending with one 'cross edge' are stored. This avoids having to repeatedly search the graph for path alternatives in the following steps.

After this structural inspection of the model, the logical dependencies between the navigation paths, which are workflows in the application, are examined by analysing the guard conditions and variable assignments on the model transitions. The resulting *dependency graph* aids the test generator in constructing a valid order of the test cases, as test cases may fulfil or block preconditions of other test cases. In the dependency graph there is a vertex for every edge in the application graph. A directed edge is inserted into the dependency graph for every dependency candidate: a transition assignment to a variable that is used in a guard of another transition. The dependencies are propagated using the application graph so that descendant edges in the application graph inherit dependencies of the predecessor edge: the intersection of dependencies for edges leading to this edge is calculated and propagated further down the graph.

For test case generation, the algorithm selects paths in the graph to be *executed*, which means that the assignments of the transitions on the path are evaluated to update the involved model variable values. For path selection the generator algorithm employs a greedy algorithm to select paths with a maximum amount of views and transitions not covered yet. The algorithm must not pick a path for execution that would block the usage of other still unvisited transitions. Using the dependency graph, paths with transitions that would be blocked by an assignment executed on a transition of the picked path can be determined. This step mostly determines the runtime complexity of the whole algorithm, as it requires partial simulation and permutation of path alternatives.

Deriving meaningful test oracles is one of the major challenges in model based testing caused by the need for abstraction [8], [9]. In the proposed approach, test oracles are generated using text expressions at the model views that may contain model variables or test data. In this way the test oracles can check for functional properties of the application that are observable in the HTML source text of the web application pages. Whenever a view is reached by a transition during test case generation, verification steps for all the text expressions of this view are created.

Another option to specify test oracles is to use the *Xpath* keyword in the view declaration (not used in the example listing). It allows to declare XPath expressions to check for structural properties of the web page returned by the server, for example, to verify that a table has the desired number of rows. In the model an XPath expression is referenced by a symbolic name to keep the model clean and abstract. The reference is resolved with the external UI mapping file to the real XPath expression (see Section *Test Case Transformation*). Furthermore, it is possible to parameterize the XPath expression with model variable expressions using a wild card that is substituted with the resolved expression. During test case generation the parameter wild card is resolved from the current model variable values. With this option one could for example check for a web page table cell containing the value of a model variable. The *Xpath* directive helps to formulate test oracles where the simple text matching based option is not applicable or not precise enough and thus makes the approach more widely usable. The effectiveness of the generated test suite largely depends on the count and quality of the *Text* and *Xpath* directives in the model that are used to generate the test oracles. More precise specification of the desired observable behaviour using these directives increases the fault finding capability of the generated test suite at the price of higher modelling and model maintenance effort.

As a means of tracing back failures detected during test execution, the used model path and input data are recorded as meta-data in the generated test cases and transferred to the test scripts in the transformation step. This allows for manual inspection of failed test cases and the identification of relevant model parts to find faulty application parts. A tool-integrated, automatic traceability function that marks the relevant model part from the test tool feedback is planned for the extension of the research prototype.

V. Test Case Transformation

To transform the generated abstract tool independent test cases into executable test scripts for a specific test automation tool, two aspects need to be considered. First, implementation details that were abstracted to ease modelling must be integrated to make the test cases executable directly on the system under test. Second, the abstract test cases must be translated into the language of the chosen test automation tool, for example, XML-based test scripts for the Canoo Webtest tool. Implementation specific data is kept in an external UI mapping file in XML format. This file serves to abstract the implementation details of the application, for example how input elements are identified. An entry in the mapping file consists of a logical reference name used in the model that maps to the reference data, for example, an XPath expression to locate a button. The Eclipse Xpand tool is used to translate the abstract test cases into the required format for the chosen test automation tool. Xpand processes the abstract test cases using a target tool specific template to generate the test scripts. The test scripts can then be executed repeatedly, for example as part of a regression test suite. The research prototype offers transformation templates for the tools Canoo Webtest and Selenium. Once adapted to a specific test automation tool, the generated test oracles perform a comparison of the given search string with the complete HTML source of the response page. To avoid repeated modelling efforts for multilingual web applications, one can replace static text fragments in the model with keys that map to files containing key-value pairs with the appropriate text for every used language. In this way, test cases for different languages can be generated from one model.

VI. Discussion

Preferably, in the targeted usage scenario the model would be developed in parallel to the web application driven by the requirements. This enables continuous cross-checking as a means to find requirement inconsistencies and application bugs in an early development stage, which is one of the advantages attributed to model-based testing approaches [7]. Although it should also be possible to reverse-engineer models at least partially from existing applications using crawling mechanisms, one would then lose the advantage of this redundancy.

In its current state, the proposed approach is suited to contribute automated test cases to a test suite where baseline scenarios are generated from models and more intricate test scenarios are designed manually by the test engineer. Due to the test oracle mechanism that relies on the comparison of expected and actual text fragments in the HTML code, it can be used to test workflows that allow such an observation of the test impact. Modern web applications increasingly employ the AJAX technology, which, currently, is not fully supported by the proposed approach. Because AJAX performs partial updates of web pages, it blurs the concept of a *View* as used in this paper. Such behaviour can be modelled in a view using variables that reflect the dynamic portions as long as the updates can be expressed as user-triggered transitions to the same view. However, due to the rising model complexity this might not be reasonable in cases where there is a high degree of dynamism. For example, in an extreme case the application could consist of just one initial webpage that is continuously altered using partial updates. Furthermore, updates that are not directly caused by user interaction, as, for example, by polling timers, can not be modelled currently.

The prototype offers further possibilities for functional enhancement. The realization of hierarchical and horizontal decomposition techniques for the application models would benefit the usability of the approach especially for larger models. Furthermore, currently only positive test cases can be generated from the model. To improve the ability to find application bugs, support for the automated generation of negative test cases is planned. For example, such test cases could be generated by the generator algorithm through selective omission of required input data at transitions. A facility to assign values to model variables that are extracted from views at runtime, e.g. using XPath expressions stored in the UI mapping file, could be added to the research prototype to support nondeterminism. The extracted values could then be used in guard expressions to guide an adaptive test case generation process.

VII. Related Work

Various methods for model-based testing of different aspects of web applications have been proposed in the literature [10]. First approaches utilized the Unified Modelling Language (UML) to model static structural aspects of the web application [11], [12]. Therefore it is not possible to automatically generate test cases for functional testing, as this would require model information about the intended dynamic behaviour of the application. These methods, however, can be used to guide the manual design of test cases.

Dynamic information is modelled for example using annotated finite state machines by Andrews et al. [2] and so called event-sequence-graphs by Linschulte and Belli [3]. These approaches offer domain specific means to model the flow of control and data. Abstract test sequences are generated by algorithmically traversing the models. In contrast to the proposed approach, the generated test sequences have to be manually supplemented with test oracle information by the test engineer and adapted to be automated with a test automation tool. Similar to the approach described in this paper, Wang et al. [4] use a directed graph to model the navigation paths of the web application. There, test sequences that cover all pairwise interactions between web pages are generated. However, it does not model the dependence of feasible navigation paths on the application state as our method does.

Ernits et al. [5] describe the usage of the open source tool *NModel* for *online testing* of web applications, which in the model based testing context means that a generated test case is executed immediately and the output of the system under test is used as input to guide the further generation of test cases from the model. The C# language is use to construct the model program. An adaptation layer contains test oracle code for both front end and back end testing, which remedies the problem of low observability of test impact pertinent to web application black box testing [2]. As interaction and verifying functionality has to be implemented in the adaptation layer by the test engineer, the authors stress that more effort is needed for implementation of this layer than for the application model itself.

Another approach to web application test case generation is to use control flow and input data that is extracted from application access logs [13]–[15]. This data is used to reconstruct user sessions, which can be seen as constructing an implicit usage model. The extracted model can therefore only reflect parts of the application that were executed by the users. In contrast to our approach, these methods can not be used to generate test cases during development before the application has been deployed. Furthermore it is not clear, how methods solely based on user session data can be used in development scenarios where the application is subject to frequent changes that require a modification of the test suite.

Marchetto et al. [16] present an approach that also utilizes recorded user interaction trace data to construct a state machine model especially for testing AJAX functionality. Input data is provided from the collected requests and test oracles are created manually. The generated test sequences are translated into the test case format of Selenium. Another approach that especially targets testing of AJAX functions of web application is presented by Mesbah and van Deursen [17]. Instead of real users providing execution traces, a special crawler is used to build a model. To provide input when necessary, the crawler is backed by a database of input data that is associated with hash values calculated from the web page form attributes. It remains unclear, how the crawler selects the correct set of input data dependent on both the web page form and the current application state. A test oracle is provided by invariant expressions on the DOM tree. Generic invariants as for example syntactic validity can be complemented by application specific invariants stated as XPath expressions.

The problem of black box testing of web applications is related to the more general problem of testing graphical user interfaces (GUI testing). Memon [18] proposed the event flow model and several strategies to automatically derive test cases from the model. Information about the state transformation of GUI components is used for test oracles. In comparison to the method proposed in this paper it does not offer a facility to express the dependence of application behaviour on test input data and does not explicitly consider the HTTP session concept. With an initial focus on test case prioritization Bryce et al. describe first efforts [19] towards a unified model and terminology for both general GUI applications and web applications building upon the event flow model and prior work based on user session data [20].

There are also approaches that use static analysis techniques of the web application source code and symbolic execution to generate test cases from the inferred control and data flow models [21], [22]. Artzi et al. [23] present a hybrid approach that also uses concrete execution. These techniques, however, target specific implementation languages and web application development frameworks. This implies that at least the analysis module must be adapted to the used implementation technologies.

VIII. Conclusion

A model based testing approach for user interface level testing of web applications was presented. A high degree of automation along the testing process is demonstrated by the research prototype, in particular the generation of test oracles from model information and the transformation of abstract test cases to executable test scripts. A case study to evaluate the efficiency in a web application development project, especially in maintenance scenarios, is planned.

Acknowledgment

This work was partially funded by Deutsche Forschungsgemeinschaft (DFG) grant BL 891/2-1.

References

[1] C. Kaner, B. Pettichord, and J. Bach, *Lessons Learned in Software Testing*. Wiley, 2001.

[2] A. A. Andrews, J. Offutt, and R. T. Alexander, “Testing web applications by modeling with fsms,” *Software and System Modeling*, vol. 4, no. 3, pp. 326–345, 2005.

[3] M. Linschulte and F. Belli, “On ’negative’ tests of web applications,” in *Proceedings of the 3rd South-East European Workshop on Formal Methods, SEEFM 2007, Thessaloniki, Greece*, 2007.

[4] W. Wang, S. Sampath, Y. Lei, and R. Kacker, “An interaction-based test sequence generation approach for testing web applications,” *High-Assurance Systems Engineering, IEEE International Symposium on*, 2008.

[5] J. Ernits, R. Roo, J. Jacky, and M. Veanes, *Testing of Software and Communication Systems*. Springer, 2009, ch. Model-Based Testing of Web Applications Using NModel, pp. 211–216.

[6] W. Grieskamp, “Multi-paradigmatic model based testing,” Microsoft, Tech. Rep., 2006.

[7] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.

[8] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” in *FOSE ’07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 85–103.

[9] A. Pretschner, M. Utting, and B. Legeard, “A taxonomy of model-based testing,” Department of Computer Science, University of Waikato, Tech. Rep., 2006.

[10] M. H. Alalfi, J. R. Cordy, and T. R. Dean, “Modelling methods for web application verification and testing: state of the art,” *Softw. Test. Verif. Reliab.*, vol. 19, pp. 265–296, 2009.

[11] J. Conallen, “Modeling web application architectures with uml,” *Commun. ACM*, vol. 42, no. 10, pp. 63–70, 1999.

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

[13] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II, “Leveraging user-session data to support web application testing,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 187–202, 2005.

[14] J. Sant, A. Souter, and L. Greenwald, “An exploration of statistical models for automated test case generation,” in *WODA ’05: Proceedings of the third international workshop on Dynamic analysis*. New York, NY, USA: ACM, 2005, pp. 1–7.

[15] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. S. Greenwald, “Applying concept analysis to user-session-based testing of web applications,” *IEEE Trans. Softw. Eng.*, vol. 33, pp. 643–658, 2007.

[16] A. Marchetto, P. Tonella, and F. Ricca, “State-based testing of ajax web applications,” in *ICST ’08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 121–130.

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

[18] A. M. Memon, “An event-flow model of gui-based applications for testing,” *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[19] R. C. Bryce, S. Sampath, and A. M. Memon, “Developing a single model and test prioritization strategies for event-driven software,” *IEEE Transactions on Software Engineering*, vol. 37, pp. 48–64, 2011.

[20] S. Sampath, R. C. Bryce, G. Viswanath, V. Kandimalla, and A. G. Koru, “Prioritizing user-session-based test cases for web applications testing,” in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 141–150.

[21] W. G. Halfond, S. Anand, and A. Orso, “Precise interface identification to improve testing and analysis of web applications,” in *ISSTA ’09: Proceedings of the eighteenth international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2009, pp. 285–296.

[22] M. Wang, J. Yuan, H. Miao, and G. Tan, “A static analysis approach for automatic generating test cases for web applications,” *Computer Science and Software Engineering, International Conference on*, vol. 2, pp. 751–754, 2008.

[23] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, “Finding bugs in web applications using dynamic test generation and explicit state model checking,” *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 474–494, 2010.