Model-Driven Testing for Web Applications using Abstract State Machines

Francesco Bolis[1], Angelo Gargantini[1], Marco Guarnieri[1], Eros Magri[1], and Lorenzo Musto[2] ⋆

[1] Dip. di Ing. dell'Informazione e Metodi Matematici, Università di Bergamo, Italy
{francesco.bolis,angelo.gargantini,marco.guarnieri,eros.magri}@unibg.it
[2] Optics Division Alcatel-Lucent, Vimercate, Italy lorenzo.musto@alcatel-lucent.com

Abstract. The increasing diffusion and importance of Web Applications has led to strict requirements in terms of continuity of the service, because their unavailability can lead to severe economic losses. Techniques to assure the quality of these applications are thus needed in order to identify in advance possible faults. Model-driven approaches to the testing of Web Applications can provide developers with a way of checking the conformance of the actual Web Application with respect to the model built from the requirements. These approaches can be used to automatically generate from the model a set of test cases satisfying certain coverage criteria, and thus can be integrated in a classical test driven development process. In this paper we present an automated technique for Web Application testing using a model-driven approach. We present a way of modeling Web Applications by Abstract State Machines (ASMs), and a process for generating automatically from the model a concrete test suite that is executed on the Web Application under test in order to check the conformance between the application and the model.

1 Introduction

The wide diffusion of Internet combined with mobile technologies has produced a significant growth in the demand of Web Applications with an increasing request for efficient techniques tailored for their validation [8]. Researchers and practitioners are still trying to find viable approaches in order to validate Web Applications. A possible approach is to apply Model-driven or model-based testing (MBT)[16] to Web Applications. Since software testing is a costly and time-consuming activity, specification-based (or model-based) testing permits to considerably reduce the testing costs. MBT consists in building an abstract model of a Web Application and using the model instead of the code to derive tests (including the oracles) and to define adequacy of the testing effort with respect to the requirements. The model of the Web Application does not need to include all the details of the implementation, but it should be precise enough to guarantee that the test cases represent actual use scenarios of the Web Application.

⋆ This work has been partially supported by the project Ricerca Applicata per il Territorio - Berg. II - Regione Lombardia and Alcatel-Lucent Spa

Having an abstract model that represents a Web Application is no longer an unrealistic hypothesis for three reasons. Firstly, several model-driven techniques for the development of Web Applications have been developed in the last decade [12] and thus by using these techniques models are easily available as a result of the development process. Secondly, techniques for extracting abstract models from existing Web Applications have been developed [2] and have been already used in several approaches to Web Application testing [1]. Thirdly, designers often manually build an abstract model from the requirements and this model is used to check whether the Web Application satisfies the requirements.

MBT can be integrated in an agile development process [14], by helping the developers to automatically derive tests and execute them. Our approach, that belongs to the third alternative presented before, assumes that initially the designer develops a model of the Web Application (in a model-driven classical view), derives some tests cases from it, and executes these tests against an empty implementation (in a test-driven development - TDD). He/she then implements the Web Application and automatically runs the tests until they pass. Any change in the code that does not require a modification of the model is checked again by the original test cases. Some modifications of the code may cause a failure of the tests because the model must be updated. In this case, the designer does not update the test cases (differently from a classical agile development) but he/she modifies the model and extracts the test cases again.

MBT can be used also in the other two scenarios presented above: (a) if the Web Application is developed by using a model-driven technique, our approach can be applied to check the correctness of the model-to-code transformations; (b) if the model is automatically built from the application, then our approach can be used to generate test suites for regression testing.

This development process works better than the classical TDD if maintaining the model, deriving the abstract test cases, and transforming them in concrete test cases is easier than maintaining the test suite. To this aim, the following features of the proposed process are critical: (1) the model must be written in a notation powerful enough to express any behavior of the Web Application, and at the same time abstract enough to ease the process of model definition; (2) it must be possible to automatically analyze and execute the models in order to find faults in them and to gain confidence that they capture the intended behavior of the application; (3) the test generation process must be automatized; (4) the concretization of the abstract tests must be automatized and the resulting tests must be automatically executed.

In this paper we propose a model-based approach to Web Application testing that uses sequential nets of ASMs and satisfies all the features listed above. Section 2 presents some background about ASMs, whereas in Section 3 we present our model based approach including a technique to model Web Applications by ASMs and how to generate and execute tests for Web Applications. In Section 4 we present an example of our approach. Section 5 presents the related work, whereas in Section 6 we draw some conclusions and present future work.

2 Background

Abstract State Machines. ASMs, whose complete presentation can be found in [5], are an extension of Finite State Machines (FSM), where unstructured control states are replaced by states with arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates defined on them. ASM states are modified by *transition relations* specified by "rules" describing the modification of the function interpretations from one state to the next one. There is a limited but powerful set of *rule constructors* that allow to express guarded actions (`if-then`), simultaneous parallel actions (`par`) or sequential actions (`seq`).

An ASM state is a set of *locations*, namely pairs (*function-name*, *list-of-parameter-values*). Locations represent the abstract ASM concept of basic object containers (memory units). Location *updates* represent the basic units of state change and they are given as assignments, each of the form $loc := v$, where loc is a location and v its new value.

Functions may be *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are distinguished between *monitored* (only read by the machine and modified by the environment), and *controlled* (read and written by the machine).

A *computation* of an ASM is a finite or infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ of states of the machine, where s_0 is an initial state and each s_{n+1} is obtained from s_n by firing the (unique) *main rule* which represents the starting point of the computation. An ASM can have more than one *initial state*. Listing 1 reports a fragment of the ASM specification of a Web Application.

```
asm index
import Pages
signature: // Declarations
    enum domain States = { EMPTY | PASSWORD | USERNAME | USERPASSW }
    enum domain Events = { SUBMIT_SUBMIT | RESET_RESET | TEXT_USERNAME |
    TEXT_PASSWORD | TEXTDEL_USERNAME | TEXTDEL_PASSWORD | LINK_PREV }
  dynamic controlled currentPage : Pages
  dynamic controlled currentState : States
  dynamic monitored event : Events
definitions: // Definitions
  macro rule r_Reset =
    event = RESET_RESET then
      if currentState != EMPTY then
        currentState := EMPTY
    endif
  macro rule r_UsernameText = ...
  ...
  main rule r_Index =
    if currentPage = INDEX then
    par
      r_Reset[]
      r_UsernameText[]
      ...
    endpar endif
default init initial_state : // Initial values
  function currentState = EMPTY
  function currentPage = INDEX
```

Listing 1: AsmetaL code of Index page

The ASM methodology has been successfully applied in different fields [5] as: definition of programming and modeling languages, modeling e-commerce and web services, design and analysis of protocols, architectural design, and verification of compilation schemes and compiler back-ends.

The ASMETA toolset [4] supports designers and developers of ASMs. They can assist the user in developing specifications and proving model correctness by checking state invariants and temporal logic properties. A number of tools helps the designer also during the validation phase by means of a model reviewer, a simulator, and a scenario-based validator. Among the ASMETA tools, ATGT is the test generation tool.

Model-based testing for ASMs. By using ASMs in MBT, we assume that a *test sequence* or *test* is a finite sequence of states, whose first element is an initial state, and each state follows the previous by applying the transition rules.

Several coverage criteria have been defined for ASMs [9]. For instance, one basic criterion is the *rule coverage* which requires that, for every rule r_i, there exists at least one state in a test sequence in which r_i fires and there exists at least a state in a test sequence in which r_i does not fire.

Starting from the definition of coverage criteria, several approaches have been defined in order to build test suites. ATGT uses a technique based on the capability of the model checker SPIN [11] to produce counterexamples [10]. A recent work [3] in this area, improves considerably the scalabilty of the approach and extends the concept of ASM to *sequential nets* of ASMs.

3 Model-based Testing for Web Application by ASM

Assuming the existence of an abstract model of the Web Application, we can use model-based techniques in order to compute an adequate test suite for the Web Application under test (AUT) in order to check whether it conforms to the model or not. We have chosen to use ASMs to represent the model because they are a good compromise between abstraction and expressive power. For instance, they are more flexible than FSMs, and they can also handle shared variables that can represent session data, which are vital for testing dynamic Web Applications.

Figure 1 shows the testing process we have devised. It takes as inputs the abstract model and the AUT. By giving the ASM model as input to the ATGT tool, we generate a set of tests according to several coverage criteria, for instance a basic criterion is the *Rule Coverage* whereas a more advanced one is the *Modified Condition Decision Coverage* [9]. The ATGT tool produces as output the Abstract Test Suite (ATS) which is a high level representation of the resulting test sequence.

Since we are testing Web Applications, we have chosen to represent our test suites in terms of the interactions that a user can do with the AUT, and thus, in order to model concrete tests and to automate their execution, we have chosen to use Sahi[3], a *capture and replay tool* that lets users express test cases using scripts

[3] Sahi Web Automation and Testing Tool - http://sahi.co.in/

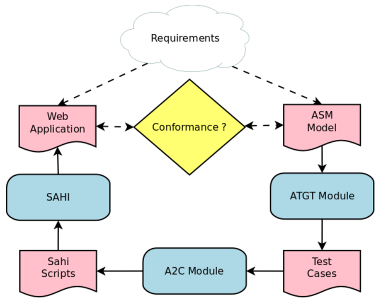

Fig. 1: Testing Process

and then it executes them. A translation is needed in order to map the ATS to a concrete set of Sahi scripts that can be executed on the Web Application under test, further details on the translation process are given below.

Then we execute the generated concrete test suite using the Sahi runner. Using the oracles automatically inserted in the scripts we can check whether the application under test conforms with the model or not.

Modeling Web Applications by using ASMs We have defined an approach that builds an ASM model for each page in the Web Application, and then the entire Web Application is represented by a net of ASMs. To be more precise, in order to avoid the problem of the combinatorial explosion in the number of states of the ASM, we have used the technique based on hierarchical decomposition presented in [3] that introduces the *sequential nets* of ASMs. The modeling of one web page is done in the following way:

1. We define a shared domain, called *Pages*, that contains a value for each page in the application and a controlled variable *currentPage* of domain *Pages*.
2. We define a domain called *States*, local to each page, that contains an adequate number of states in order to represent the behavior of the page. We define a controlled variable *currentState* of domain *States*.
3. For each input element e in the web page and for each event associated with e, we define a constant in the *Events* enumerative domain. Each constant identifier reflects the kind and name of the event and the input element. We define a monitored variable *event* of domain *Events*. The kinds of the input elements are: (a) links, (b) buttons, (c) submit buttons, (d) reset buttons, (e) text fields and text areas, (f) password fields, (g) file dialogs, (h) checkboxes, (i) radio buttons, and (j) select lists.
4. For each input element e we write a rule that handles the events associated with e by either updating the current state of the page or executing the transition to another page, represented by another ASM.

A2C: Translating Abstract Tests to Concrete Tests The A2C module is a general purpose translator from an abstract test sequence into a concrete script in a selected scripting language. The application scans the abstract test sequence in

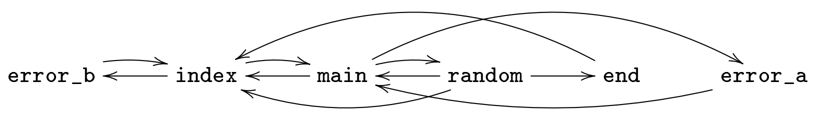

Fig. 2: Web-based application case study

order to extract the value of the *event* variable and it generates the concrete script according to this value. The module uses a template in order to describe the rules that guide the translation process; in this way if we want to extend the A2C tool in order to support a particular scripting language we only have to define a new template file. For instance, the Sahi transformation rule for a click event is as follows:

SUBMIT(name) ::= «**_click**(**_submit**("*name*"));»

4 Case Study

As a case study we have chosen a simple PHP Web Application, already presented in [13], that is composed by six different pages:

index.php is the login and initial page that requires to the user the username and the password in order to access to the other pages. It contains a *Reset* button and a *Submit* button that opens up the **main.php** page.

error_b.php is an error page opened up if any information in **index.php** is missing or wrong.

main.php contains several input elements, such as a link, a file dialog that can be activated by clicking on the *Browse* button, a textbox, a checkbox and the button *Submit* that activates the **random.php** page.

error_a.php is an error page opened if any field in **main.php** is missing.

random.php contains two link, one to **index.php** and the other one to **main.php**, a drop-down menu, radio buttons and a *Submit* button that activates the **end.php** page.

end.php contains a link to **index.php** or the option of closing the browser.

By following the process presented in Section 3, we have modeled each page using an ASM and then we have built a sequential net of ASMs, which is shown in Figure 2. Using the technique presented in [3] allowed us to avoid the explosion in the number of states in the model.

Once we have defined the model, we have used it in order to generate the test cases, which have been translated in Sahi scripts. Listing 2 shows a snippet of an ATS generated by *ATGT* from the ASM presented in Listing 2, whereas Listing 3 shows the corresponding Sahi scripts generated by the *A2C* tool.

ATGT has generated 212 test cases that achieved a 100% coverage of the PHP source code. We have used *XDebug*[4] and *PHP-Coverage*[5] in order to compute the coverage achieved by the test suite.

[4] XDebug, Debugger and Profiler Tool for PHP - http://xdebug.org/

[5] PHPCoverage, code coverage tool for PHP - http://phpcoverage.sourceforge.net/

```
[ currentState=EMPTY
  currentPage=INDEX
  event=TEXT_USERNAME ]
[ currentState=USERNAME
  event=TEXT_PASSWORD ]
[ currentState=USERPASSW
  event=SUBMIT_SUBMIT ]
[ currentState=EMPTY
  currentPage=MAIN ]
```

Listing 2: ATS Example

```
_navigateTo("index.php");

_setValue(_textbox("username"),"admin");

_setValue(_textbox("password"),"admin");

_click(_submit("submit"));

_assertEqual("main.php",top.location.href);
```

Listing 3: CTS Example

5 Related Work

An approach quite similar to ours is the one presented by Andrews et al. [2]. They developed *FSMWeb*, a tool that can be used to test Web Applications. They model the Web Application by a hierarchy of FSMs, where a FSM represents either a logical web page, i.e. the model of a subsystem of the AUT, or a top level FSM, i.e. an aggregate of logical pages. In our opinion modeling Web Applications with ASMs offers a higher degree of expressiveness w.r.t. the FSMs. They propose also a way of automating the definition of the model from the Web Application under test. However, their tool does not implement this feature and thus they are tied to a manual implementation of the model as in our approach.

Deutsch at al. [7] present an approach that models data-driven Web Applications by means of ASM$^+$ models, which represents the transitions between pages, determined by the input provided to the application. Our approach can be applied to a wider range of Web Applications, i.e. actually it works with any Web Application for which exists an ASM model. In our opinion handling the testing of events by linear or branching-time temporal logics leads to complex models that can made the integration with agile development too hard, although it can discover more subtle errors. Another advantage of our approach is that we can use all the features provided by the ASMETA tool set, including a simulator, a model checker and a model advisor.

Tonella and Ricca [15] propose a technique to automatically generate and execute test cases starting from a UML model of the Web Application. Their approach requires a manual intervention in several phases, i.e. in the UML modeling phase and in the test refinement phase (their tool requires that the user fills in the input values in each URL), whereas our approach requires the intervention of the user only in the model definition phase. This is an advantage primarily because if we are in a situation in which the model already exists, the testing process can be executed without any intervention from the user.

6 Conclusion and Future Work

We have presented our ongoing work on using MBT for Web Application in an Agile context. Our approach provides the designers with an expressive but abstract language, an automatic generator of tests and a translator to concrete tests, and an automatic executor of the tests over the AUT.

A crucial activity in the application of our approach, is building an abstract model of the AUT. We plan to provide some help during this phase by generating automatically part of the ASM model from the AUT. We also plan to extend our approach with a model-to-model transformation tool which takes as input WebML [6] models, i.e. only Navigation and Composition models, and translates them into ASMs. Given the fact that WebML is a well-known Web application modeling language, its use could ease the definition of ASMs.

We also plan to study how our approach behaves on a real Web Application, i.e. the web interface ZIC (Zero-Installation Craft) integrated in some of the Alcatel-Lucent devices.

References

1. M. Alalfi, J. Cordy, and T. Dean. Modelling methods for web application verification and testing: state of the art. *Softw. Test, Verif. Reliab*, 19(4), 2009.
2. A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4:326–345, 2005.
3. P. Arcaini, F. Bolis, and A. Gargantini. Test generation for sequential nets of abstract state machines. In *Proc. of ABZ*, number 7316 in LNCS, 2012.
4. P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Softw., Pract. Exper.*, 41(2):155–166, 2011.
5. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer, 2003.
6. S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (WebML): a modeling language for designing web sites. 2000.
7. A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *J. Comput. Syst. Sci.*, 73(3), 2007.
8. G.A. Di Lucca and A.R. Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12), 2006.
9. A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science*, 2001.
10. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *Proc. of ASM*, number 2589 in LNCS, 2003.
11. G. Holzmann. *Spin model checker, the: primer and reference manual.* Addison-Wesley, 2003.
12. I. Manolescu, M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. Model-driven design and deployment of service-enabled web applications. *ACM Trans. Internet Technol.*, 5(3), 2005.
13. A. Memon and O. Akinmade. Automated Model-Based Testing of Web Applications. In *Google Test Automation Conference 2008*, 2008.
14. O. Puolitaival. *Adapting model-based testing to agile context.* VTT, 2008.
15. F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proc. of ICSE.* IEEE, 2001.
16. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan-Kaufmann, 2006.