Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1311–1314

ISBN 978-83-60810-51-4

Formal specification to support advanced model based testing

Karel Frajták, Miroslav Bureš, Ivan Jelínek
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13, 121 35 Praha 2, Czech Republic
Email: {frajtak, buresm3, jelinek}@fel.cvut.cz

***Abstract*—Reliability and correctness of a web application are crucial factors to its success. Errors occurring in non-deterministic moments do not attract the target audience of the application. It is quite impossible to deliver 100% reliable application. Hidden errors are discovered when target users are using the application. With proper tooling and support the time between the error discovery or report and the error elimination can be reduced. In this paper we are proposing formal model for a model based improvement of a testing process of the web application. Our goal is to create a formal model and design a testing framework based on the direct guidance of a tester through the testing process, verifying his steps and providing better feedback.**

***Index Terms*—test automation, test case scenario generation, guidance application.**

I. Introduction

In a software development life-cycle a software testing phase is the most important phase. In this phase most errors are discovered and fixed. Different types of software testing techniques can be chosen to be used in the testing phase. For example unit tests are executed automatically and repeatedly to discover the errors made in development phase when a new feature was introduced. Only single subsystem or individual module is tested using unit testing approach. The more subsystems are involved in unit testing, the more complicated the test set up and the initialization phase are—the subsystems that are not directly under test must be mocked or proxies have to be created to simulate their functionality.

Logical next step is integration testing approach where individual modules are combined and tested as a group. As testing becomes more complex, a different approach must be taken. The steps in integration tests are described in test case scenarios. For every step in the test case scenario a action that have to be taken and set of the input values to be supplied is defined.

Test case scenarios created manually by test coordinators are usually given to testers. Testers have to follow the steps specified in these test scenarios and testewr reports back the result of the testing process. The main problem is the quality of provided feedback, untrustworthy tester can report no problems at all.

An architecture of our proposed system was presented in [3]. The proposed system guides the tester through every single step of test scenario. Every input value and every action taken by the tester is verified. One of the important parts of the proposed system is the test scenario dispatcher. It watches the entire system under test (SUT) and serves the scenarios to the tester based on various criteria—for example it dispatches the scenarios for testing of mission critical subsystems first, or do not dispatch tests for subsystems currently not available.

In our proposal we will focus on a model based improvement of a testing process of the web application. It will be achieved by eliminating the unnecessary paperwork by introducing the formal automatic test case specification. Our application for tester's guidance proposed in our previous paper [3] will help tester through the testing process. The guidance application that will be designed, is based on the proposed formal model.

II. Related work

In [1] web application is described as a set of web page schemas. User interaction with the web page schema is described by input and action schemas. On a single page represented by a web page schema user input is captured in a input schema. Then the user executes one of the actions available in the action schema. The resulting sequence of events is verified after. We have adapted the definition of the web application published in [1] to suit our needs. We need to capture business properties, resp. technical properties not directly related to the problem domain and we call them business, resp. technical (non-model) metadata. Maximal throughput of given segment of the web application is an example of a technical metadata. Metadata will be used in a test case generation and during the test case scenario dispatching.

It is not possible for humans to verify every web page schema and every possible combination of input values and actions taken. Testing of the system based on its model, so called model-based testing, is the key approach here [2, 4, 5]. We will generate these tests automatically. Model of the web application is input to the test case scenario generator.

The application can be modelled using UML [6] notation. In [5] test case scenarios are generated using UML activity diagrams. These diagrams are used to express all possible controls flows in defined uses cases. Many aspects of the web application can be expressed in UML diagrams, but there are some limitations. For example the relationship between web pages, their input and action elements. WebML [7], an UML–like graphical notation, allows the architect to describe these features.

For the design of the solution a methodology called Test by Contract [8] can be used. It defines test contracts extending the definition of the component contracts concept [4] to the test domain. A component contracts specify how to use the component interfaces correctly to access component functionality. The component contracts capture the mutual responsibilities that both partners of a component (i.e. service contractor and client) must comply with, independent of how they are implemented. Any violation of these contracts indicates a potential fault.

The test contracts are designed and constructed based on the well-developed component contracts (e.g. interface contracts, element contracts, etc.), and are used in various testing approached (unit testing, integration testing, etc.). Their implementation using the form of aspects (precondition, post condition and invariant) results in designing component test cases based on contracts - contract-based component test. Operations for testing of the system components are going to be essential part of the system and they will allow dynamic system testing

III. Formal model

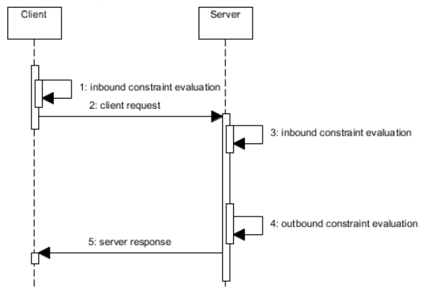
System under test, a web application, is based on the following formal model. We have adapted the formal definition of the web application published in [1]. We have extended this definition with definition of finite sets of metadata (technical and business) values.

Definition 1: A web application $\mathcal{A}$ is a tuple $\langle W, S, I, A, W_0, W_\varepsilon, M, T\rangle$, where:

- S, I, A are disjoint finite sets of state, input, and action values. Constant values can be shared among these sets.
- W is finite set of the representations of web pages,
- $W_0 \in W$ is the representation of home page, $W_\varepsilon \notin W$ is the the representation of error page,
- M is a set of business metadata values,
- T is a set of technical (non-model) metadata values

In the testing process tester has to pass through all the test case scenario steps defined in selected test case scenario. A specification of an interaction of the tester with system under test is defined in each of test case scenario step. Instructions describing the test case scenario step are defined clearly without ambiguities. For brevity we will limit the user interaction with the SUT to only one action that is captured in the test scenario step.

Definition 2: Test case scenario is finite sequence $\{\langle V_i, S_i, I_i, A_i, R_i, TD_i\rangle\}_{i\geq 0}$ where V_i is a web page page representation used in step i, $S_i \subset S$ is set of state values used in step i, $I_i \subset I$ is set of values of input elements used in step i, $A_i \in A$ is an action taken in step i, R_i is set of restricting constraints rules, $TD_i \subset S_i \cup I_i \cup M \cup T$ is finite set of state, input, business and technical metadata values used in step i. $\langle V_i, S_i, I_i, A_i, R_i, TD_i\rangle$ is called configuration of test case scenario step.


Client
Server
1: inbound constraint evaluation
2: client request
3: inbound constraint evaluation
4: outbound constraint evaluation
5: server response
Fig. 1. Locations of constraint evaluation

The tester will be therefore instructed what to do, where to do it, how the data will be shaped and under what conditions it will be done. One step in test case scenario is described by:

- web page that uniquely identifies the "location" of user interaction (where)
- action to be taken (what to execute)
- constraint rules (under what conditions)
- step data (domain entities, domain metadata, technical metadata) that will be used to evaluate constraint rules (shape of data)

If any constraint is violated, the user has option to fix the problem—user can enter the right input values or can abort the test case scenario and create an error report and provide useful feedback. If an unexpected error occurs in the application (this often indicates a problem inside the system) the test case scenario is automatically aborted. Discovered problem is reported back to the developers with the context of the SUT containing all data necessary to reproduce the error and fix it. Tester can provide additional feedback before the error report is submitted.

In standard client-server web application client requests a page, the server receives the request, executes corresponding code and sends back response. This response consists only of data—HTML, images or data. We try to capture the user interaction with the SUT in test case scenario step. The user interaction with SUT is bounded by client request and server response, either synchronous or asynchronous.

The constraint rules can be therefore divided into two separate groups: inbound constraint rules and outbound constraint rules. See Figure 1 for a schema describing the location of invocation of constraint evaluation.

The inbound client-side constraint rules are rules validating the user input before it is sent to the server side and a violation of such rule can be signalled immediately to the user and no request is raised—a server round trip is saved. The client-side inbound constraint rules are validating elements from I_i. In a similar way, the inbound server-side constraint rules are the rules validating user input, server-side state, any metadata and database right before the server action is executed. The inbound client-side constraint rules can be implemented using client-side scripting language running in the browser, for example JavaScript.

Definition 3: Inbound client-side constraint rule is a function $\phi_{in} : I \rightarrow \{0, 1\}$.

Definition 4: Inbound server-side constraint rule is a function $\Phi_{in} : S \cup I \cup M \cup T \rightarrow \{0, 1\}$.

The outbound server-side constraint rules are the rules validating the state on the server side, any metadata and database right after the server action is executed and before the data is sent back to client.

Definition 5: Outbound server-side constraint rule is a function $\Phi_{out} : S \cup I \cup M \cup T \rightarrow \{0, 1\}$.

Example 3.1: The human-readable instruction in test case scenario instructs the tester to *"Login into the system as a user with user name 'administrator'."*. The constraint rule expression *"User name is equal to 'administrator'"* can be of both types—either client-side or server-side inbound constraint rule.

The configuration for the i-th test case scenario step from Example 3.1 can be described as the following:

- web page $V_i = \{loginpage\}$
- action $A_i = \{login\}$
- constraint rules:
  - client-side inbound constraint rules: value of the input element identified by name `UserName` is 'administrator', although for simplicity this can be specified as *"User name name is equal to 'administrator'"* and expressed in suitable constraint language (for example object constraint language, OCL) in following form: `UserName` == 'administrator'
  - $R_i = \{UserName == \text{`}administrator'\}$
- data - values of input elements identified by `UserName` and `Password` identifiers
  - $TD_i = \{value_{UserName}, value_{Password}\}$

Example 3.2: On the contrary the instruction *"Login into the system as a user with administrator rights."* can be defined only as a server-side outbound constraint rule, because the system cannot verify whether the user logged in has administrator rights until the moment data is ready to be sent back to client.

The configuration for the i-th test case scenario step from Example 3.2 can be described as the following:

- web page $V_i = \{loginpage\}$
- action $A_i = \{login\}$
- constraint rules:
  - server-side outbound constraint rules: User is in administrator role
  - $R_i = \{\text{User is in administrator role}\}$
- data: empty set—we don't care about input values in this step, but input values set to the server in the request can be used for tracing and other purposes
  - $TD_i = \emptyset$

The form on login page is very simple—it contains only two input elements (`UserName`, `Password`) and two action buttons (`Login`, `Register`). The tester can locate the elements specified in the constraint rules very easily in this case, but can have problems with more complicated web forms with tens of input and action elements. We will introduce some usability features to make the tester interaction with our guideline application and SUT more comfortable. For example the name of the element in constraint rule can navigate the tester in the form and locate and highlight the corresponding input element.

The server-side inbound, resp. outbound, constraint rules are executed before, resp. after, the action is executed. The code of the system itself remains unchanged. We will encapsulate the execution of the action in our execution context. In many web frameworks for developing application these point-cuts can be implemented using filters, for example before, after, and "around" filters in Ruby on Rails. For other web frameworks not supporting this features an aspect oriented framework should be used to implement this.

Definition 6: Test case scenario step i with configuration $\langle V_i, S_i, I_i, A_i, R_i, TD_i\rangle$ where V_i is a web page page representation used in step i, $S_i \subset S$ is set of state values used in step i, $I_i \subset I$ is set of values of input elements used in step i, $A_i \in A$ is an action taken in step i, R_i is set of restricting constraints rules, $TD_i \subset S_i \cup I_i \cup M \cup T$ is finite set of state, input, business and technical metadata values used in step i, is called passed when for all rule constraints $f \in R_i$

- if f is client-side inbound rule, then $f(j) = 1$ where $j \in I_i$
- if f is client-side outbound rule or server-side inbound rule, then $f(k) = 1$ where $k \in TD_i$
- V_{i+1} is not error web page W_ϵ
- execution of action A_i results in success

Definition 7: Let $\mathcal{W} = \langle W, S, I, A, W_0, W_\varepsilon, M, T\rangle$ be a web application. A test scenario of W is an finite sequence of test case scenario step configurations $\{\langle V_i, S_i, I_i, A_i, R_i, TD_i\rangle\}_{i\geq 0}$ where V_i is a web page page representation used in step i, $S_i \subset S$ is set of state values used in step i, $I_i \subset I$ is set of values of input elements used in step i, $A_i \in A$ is an action taken in step i, R_i is set of restricting constraints rules, $TD_i \subset S_i \cup I_i \cup M \cup T$ is finite set of state, input, business and technical metadata values used in step i, we say that given test case scenario is passed when each test scenario step K_i is passed for each $i \geq 0$.

Our target is to find errors left in the system that were not discovered by the unit testing. To have higher confidence about the correctness of the implementation of the system the SUT must be covered by our test case scenarios—every single action the user can execute must be used in at least one test case scenario step and this test case scenario must be successfully completed by the tester without problems.

Definition 8: A test scenario step K_i with test case scenario step configuration $\langle V_i, S_i, I_i, A_i, R_i, TD_i\rangle$ where V_i is a web page page representation used in step i, $S_i \subset S$ is set of state values used in step i, $I_i \subset I$ is set of values of input elements used in step i, $A_i \in A$ is an action taken in step i, R_i is set of restricting constraints rules, $TD_i \subset S_i \cup I_i \cup M \cup T$ is finite set of state, input, business and technical metadata values used in step i, then action A_i, resp. input element I_i, resp. action A_i, is covered by K_i when K_i is passed.

Definition 9: Given the finite set A of action symbols of the web application $\mathcal{A}$, then

$$a_{verify} = \frac{|A_{verified}|}{|A|}$$

is called test case action verification ratio of the SUT

$$A_{verified} = \bigcup_{T \in E} \bigcup_{\tau \in T} \{a | a \in A_i \wedge \tau \textit{ is passed}\}$$

is the finite set of actions covered by all passed test scenarios. E is the set of executed test case scenarios, T denotes test case scenario test, τ is i-th single step in test case scenario with configuration $\langle V_i, S_i, I_i, A_i, R_i, TD_i \rangle$.

We then use $R_a = 100\% \cdot a_{verify}$ to express the SUT test case action verification percentage. The test case state, respectively input, verification ratio of the SUT are defined in a similar way.

Definition 10: Given the finite set S of state symbols of the web application $\mathcal{A}$, then

$$s_{verify} = \frac{|S_{verified}|}{|S|}$$

is called test case state verification ratio of the SUT and

$$S_{verified} = \bigcup_{T \in E} \bigcup_{\tau \in T} \{s | s \in S_i \wedge \tau \textit{ is passed}\}$$

is the finite set of state covered by all passed test scenarios. E, T and τ are defined as in Defintion 9.

Definition 11: Given the finite set I of input symbols of the web application $\mathcal{A}$, then

$$i_{verify} = \frac{|I_{verified}|}{|I|}$$

is called test case input verification ratio of the SUT, and

$$I_{verified} = \bigcup_{T \in E} \bigcup_{\tau \in T} \{i | i \in I_i \wedge \tau \textit{ is passed}\}$$

is the finite set of input constants covered by all passed test scenarios. E, T and τ are defined as in Defintion 9.

Our goal is to achieve nearly 100% test case verification of the SUT. Since it is possible for the test designers to create test case scenarios with steps covering every possible action in system manually, this can be cumbersome for them to use our approach. Existing model of the SUT describing the system can be used to automatic test case scenario generation. Test case scenario steps covering every action in the system will be generated and test designer can customize them and combine them to build meaningful test case scenarios.

IV. Conclusion and future work

In this paper we have presented a formal model for a model based improvement of testing process of the web application. Our goal was to create a formal model and design a testing framework based on direct guidance of a tester through testing process. Our application for the tester's guidance proposed in [3] helps tester through the testing process.

Our solution is based on an automatic test case scenario generation. The automation of this process will help to generate the test case scenarios effectively covering the system under test providing a higher test coverage. Model of web application formalized by our proposed formal model will be used as an input to this model based test case scenario generation.

We have adapted formal model of web application published in [1] and extended it with the notation of technical, resp. business, metadata. The metadata represent properties not directly related to the problem domain. It will be used in test case generation and during the test dispatching.

The key component of our proposal is the guidance application which will be implemented. By using this application we will eliminate the unnecessary paperwork - no test case scenarios will be created manually, test specification documents will be created with assistance of our system and the feedback provided by the tester on every step taken will be provided through our system. We focus on improving tester's feedback and verifying tester's work and validating all steps of test case scenario taken.

The future work will include an implementation of the application for tester's guidance. We will develop the extension points that will be added to SUT and will be used to communicate with guidance application and the backend system.

References

[1] Alin Deutsch, Liying Sui, and Victor Vianu. "Specification and verification of data-driven Web applications". In: *J. Comput. Syst. Sci.* 73.3 (May 2007), pp. 442–474.

[2] Mohamed El-Attar and James Miller. "Developing comprehensive acceptance tests from use cases and robustness diagrams". In: *Requir. Eng.* 15.3 (Sept. 2010), pp. 285–306. ISSN: 0947-3602.

[3] Karel Frajták, Miroslav Bureš, and Ivan Jelínek. "Manual testing of web software systems supported by direct guidance of the tester based on design model". In: *World Academy of Science, Engineering and Technology*. Paris, FR, 2011, pp. 542–545.

[4] Helaine Sousa et al. "Building Test Cases through Model Driven Engineering". In: *Innovations in Computing Sciences and Software Engineering*. Ed. by SousaEditors. Springer Netherlands, 2010, pp. 395–401.

[5] T.T.D. Trong. *Rules for Generating Code from UML Collaboration Diagrams and Activity Diagrams*. Colorado State University, 2003.

[6] *Unified Modelling Language*. Object Management Group. URL: http://www.uml.org.

[7] *Web Modelling Language*. URL: http://www.webml.org.

[8] Weiqun Zheng and Gary Bundell. "Model-Based Software Component Testing: A UML-Based Approach". In: *6th IEEEACIS International Conference on Computer and Information Science ICIS 2007*. 2007, pp. 891–898.