An Approach for Generating Concrete Test Cases Utilizing Formal Specifications of Web Applications

Khusbu Bubna

***Abstract*—As web applications are becoming more and more ubiquitous, modeling and testing web applications correctly is becoming necessary. This paper suggests utilizing two formal specification languages, State Chart and Z notation to describe the various functional requirements of web applications. The State chart framework is used to model the navigation behavior, while the Z notation is used to model the business logic functionality of web applications. Abstract test cases are generated from these two formal specification languages. An approach to map the abstract test cases generated from State chart model to Selenium RC JUnit concrete test case is presented in this paper. An example web application, a Hospital Management System is used to illustrate our approach.**

***Index Terms*—Web applications, testing, state chart, Z notation.**

I. Introduction

Nowadays, the use of web applications has grown to a huge extent. Web applications are used in almost any conceivable application, online shopping, banking, social media, etc. So, designing and testing web applications have become very crucial. Formal specification languages are used to clarify customer's requirements by helping to remove ambiguity, inconsistency and incompleteness in the software requirements and software design. Thus, they are helpful in reducing the requirement errors in the early stages of the software development life cycle. The disadvantage of formal specifications is that the cost of the specification phase of the software development life cycle increases as more time and effort are needed in developing the specifications. Software testing accounts for more than 50% of the total software development cost. If test cases can be derived directly from formal specifications, a huge reduction in the testing cost can be achieved.

In this paper, we have used two formal specification languages: State charts, which is a finite state based language to model the navigation behavior aspect of our web application, and a model based formal specification language, Z notation to model the functionality or business logic of our web application. We derive abstract test cases for testing the functionality of web applications from these two formal specification languages. The abstract test cases derived from the formal models were on the same level of abstraction as the model, and hence could not be directly executed on the implementation. In this paper, we have given a mapping between phrases used in the state chart model to Selenium RC Junit concrete test cases suitable for execution on the implementation. The remainder of the paper is organized as follows. Section II gives a brief review of related work. Section III explains our proposed methodology. Section IV describes the correspondence between state chart model and Z model, whereas Section V explains the test generation methodology using the two models. Section VI concludes the paper and offers scope for future work.

Manuscript received November 25, 2015; revised March 21, 2016.

Khusbu Bubna is with International Institute of Information Technology, Bangalore, India (e-mail: khusbu.bubna@iiitb.org).

II. Related Work

A number of formal, informal and semi-formal models like automata [1], state chart [2], UML and OCL [3], UML based web engineering, alloy, directed graph and control flow graphs, SDL, term rewriting systems have been proposed in literatures for modeling web applications. A methodology for the generation of concrete executable tests from abstract test cases using a test automation language, Structured Test Automation Language (STAL) was proposed in [4]. The authors in [3] have proposed UML class diagram, and [2] proposes state charts for modeling web navigation. The work in [5] uses Object Z to describe functional requirements of web applications and an approach is proposed to generate test sequences from the model. Object Z and state charts have been combined (specification language OZS) [6]. The objective was to extend the expressive capabilities of Z with behavioral features.

III. Proposed Methodology

There are two main components of any web application system: the front end which involves reactive navigation behavior (traversal through static and dynamically generated web pages), and the back end functionality or business logic which involves data modeling and transformation. In this paper, instead of describing the whole system in one specification formalism, we have used two different formalisms to specify different aspects of the system, thus gaining the advantages and avoiding the disadvantages of each particular formalism. State charts are used to describe the reactive navigation behavior, because of their ability to clearly describe the states and systems reaction. For the data transformation, we use the Z specification language because of its mathematical like notation and expressiveness. Compared to the formalisms for describing reactive systems (like state charts), the Z notation appears quite weak to describe liveliness (reachability of certain states) and timing constraints. Whereas the state chart formalism provides only limited support for description of data and data transformations (functional view). We also describe the correspondence between the two models in Section IV.

A. Formal Web Navigation Model: State Chart

From among the various finite state based formal specification languages, we have used State charts for modeling the navigation behavior of our web application. Each web page of our application is modeled as a separate state in the state chart model. Fig. 1 shows the state chart model of the case study of Hospital Management System. When the user navigates from one page to another, there is a transition between the corresponding states. A transition is labelled by *<event>[<guard>]/<action>*. Only the *<event>* is mandatory.

B. Formal Web Business Logic Model: Z

Web business logic specifies the web application's functionality by specifying the detailed processing done in the functional requirements as well as the constraints in the application. From among the model based formal specification languages, we have used Z notation for modeling the business logic of our application as given in the Appendix.

For example of our Hospital Management system, in the Z specification, [PATIENT] describes the set of all Patients. RESPONSE can take values either *patientLoginSuccess*, *patientLoginFailure*, *patientRegistrationSuccess*, or *patientRegistrationFailure*. *Hospital* is defined as a schema in which variables *loggedInPatients* and *registeredPatients* are defined to be sets of PATIENT with the invariant that *loggedInPatients* is a subset of *registeredPatients*. *PatientRegisterSuccess* is a schema operation in which patient is taken as an input (symbol?) of type PATIENT and *response* is an output (symbol!) of type RESPONSE. The declaration Δ*Hospital* indicates that the operation is describing a state change. In our system, it implies that *Hospital* log is updated. The pre-condition for this operation is the input patient should not belong to the set of *registeredPatients*. As described in the schema, *registeredPatients* is updated to add the new patient, and the corresponding output response is *patientRegistrationSuccess*.

IV. Correspondence between State Chart Model and Z Model

The front end navigation from one web page to another as specified in the state chart model has a corresponding back end business logic operation in the Z specification, as given in Table I for the case study of Hospital Management System. In the navigation state chart model, the transition from state Patient Login web page to state Patient Dashboard web page corresponds to the Z operation PatientLoginSucess. When the transition from state Patient Login web page to state Patient Dashboard web page takes place in the state chart model, the Z specification specifies the corresponding business logic that would take place, i.e. if patient is not logged in, then loggedInPatients will be updated after the patient logs in. The Z specification is also useful in providing details to the database designer about kinds of database query operation that should be used in the implementation. For example, in the schema operation PatientLoginSuccess in the Z specification, the statement

$$loggedInPatients' = loggedInPatients \cup \{patient?\}$$

gives information to the database designer that an update query is needed to be used in the implementation of the Hospital Management System.

TABLE I. Correspondence between State Chart Model and Z Model

Transition in state chart		**Corresponding Z operation**
Previous state	**Next state**	
Patient_Login	Patient_Dashboard	PatientLoginSuccess
Patient_Login	Error_Page	PatientLoginFailure
Patient_Reg	Patient_Login	PatientRegisterSuccess
Patient_Reg	Error_Page	PatientRegisterFailure

V. Test Generation

A. Test Generation from State Chart Model

In our approach, we have used model checking to generate test cases from state chart model. Model checking is a formal verification technique which is used to determine whether a system model satisfies certain properties. But model checking can also be used to generate test cases, and is one of the ways of doing model based testing. Abstract test cases can be generated by formulating a temporal logic specification as a trap property to be verified. Trap property is the negation of the original temporal logic specification. A counter example is generated if the model does not satisfy the temporal logic formula. A counter example is an execution trace that will take the model from its initial state to a state where the violation occurs. Finally, an abstract test case can be generated from the counter example. The state chart navigation model was transformed into an SMV program, and the trap properties into CTL formulas. Then, the Symbolic Model Verifier (NuSMV) tool was run which generated the counter examples.

In our state chart model, the state *Patient_Dashboard* was reachable from the initial state in the state chart. So, the CTL Specification property for specifying that *Patient_Dashboard* is reachable from the initial state is written as *EF(state=Patient_Dashboard),* which was negated to generate trap property as specified below.

CTL Trap Specification
Property: !*EF*(*state=Patient_Dashboard*)

This trap property will generate a counter example which is our abstract test case.

The abstract test cases generated from the state chart model are on the same level of abstraction as the formal specification model. The generated test cases are abstract because the state chart model it was generated from contained only partial information of the implementation under test, and thus cannot directly execute on the implementation. An executable test suite is needed for the system under test. This is achieved by mapping the abstract test cases to concrete executable test cases suitable for execution on the implementation. Selenium is a browser automation testing tool that is used for automating the web based applications. Selenium Remote Control (RC) is used to write automated web application UI tests. In our approach, we have proposed a mapping between phrases used in the state chart model to

Selenium Remote Control Junit test code which will help to translate an abstract test to concrete Selenium RC JUnit test code. These mappings were created manually. Fig. 2 shows the proposed test generation method from state chart web navigation model. Fig. 3 shows the Selenium RC Junit java code for Patient Registration Page of a Hospital Management System. Fig. 4 shows the xml file which gives a mapping between phrases used in the state chart model to Selenium RC Junit java code for the case study of Hospital Management System for the Patient Registration page. For example, the phrase *username1* used in the state chart model in Fig. 1 is mapped to the Selenium RC Junit Java code *selenium.type("//input[@name='username']",u1)*; as shown in the mappings xml file in Fig. 4. Concrete Selenium RC JUnit executable tests can be derived using these mappings from the abstract tests generated from the state chart model.

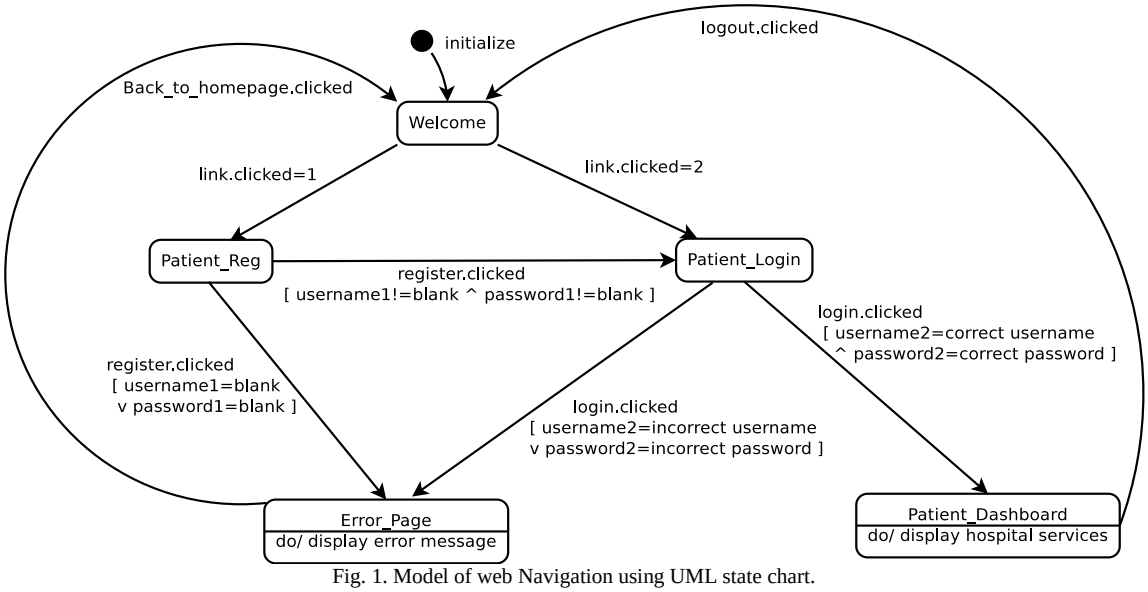

Fig. 1. Model of web Navigation using UML state chart.

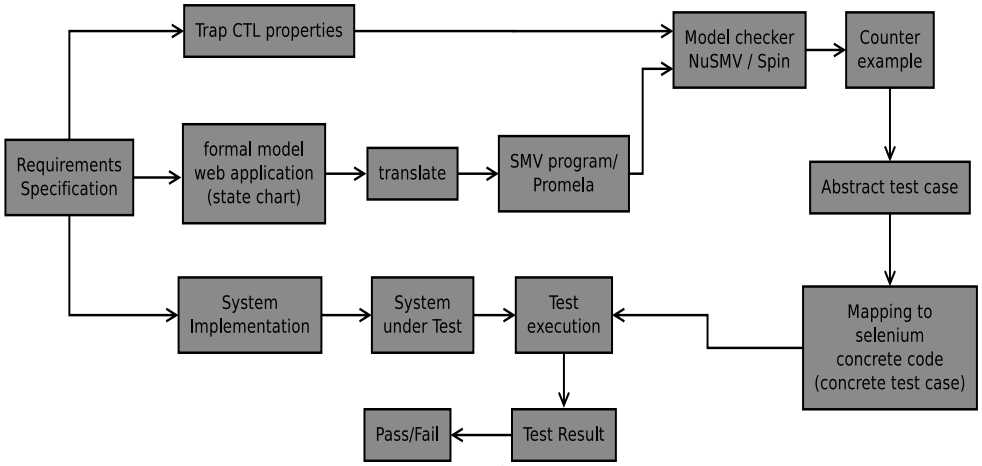

Fig. 2. Proposed test generation method from state chart web navigation model.

While converting these abstract test cases to concrete test cases, it was necessary to add information for inputs that were abstracted out from the model. From the source code implementation of the case study of Hospital Management System, we identified additional input fields which were absent in the state chart model. The additional input fields in the implementation of the Patient Registration page (*Patient_Reg* in the state chart model) were *Name* and *Age*. For these input field variables, the corresponding lines in the Selenium RC Junit java code were identified and were included in the mappings xml file. The domain of the input values of these input field variables were also identified from the implementation and were included in the mapping xml file.

The phrases used in the model may appear multiple times in different abstract test cases. The advantage of creating the mapping between phrases to Selenium RC JUnit code is the ability to reuse Selenium JUnit code next time the same phrase appears in another abstract test case (counter example in the case of state charts). For example, if the phrase *username1* used in the state chart model appears in another counter example, then the corresponding Selenium Remote Control JUnit code given in the mappings xml file can be reused. The mapping between phrases used in the state chart model to Selenium RC JUnit java code were currently done manually by observing the Selenium RC JUnit code and the state chart model. But we are trying to bring in automation in the future.

B. Test Generation from Z Model

We used Fastest, a model based testing tool to generate abstract test cases from the Z specifications. These abstract test cases were also converted to concrete test cases which have been omitted in this paper due to space constraints.

```
import com.thoughtworks.selenium.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
import java.util.regex.Pattern;
public class SelTest1
{
    private Selenium selenium;
    @Before
    public void setUp() throws Exception
    {
        selenium = new DefaultSelenium("localhost", 4444, "*firefox
        /Applications/" + "Firefox.app/Contents/MacOS/firefox-bin",
        "http://localhost:8089/");
        selenium.start();
        selenium.open("http://localhost:8089/Jkek/PatientRegistrationPage
        ");
    }
    @Test
    public void testUntitled() throws Exception
    {
        Thread.sleep(2000);
        selenium.type("//input[@name='name']", "abc");
        selenium.type("//input[@name='age']", "8");
        Thread.sleep(2000);
        selenium.type("//input[@name='username']", "abc");
        selenium.type("//input[@name='password']", "abc");
        Thread.sleep(2000);
        selenium.click("//input[@name='submit']");
        selenium.waitForPageToLoad("30000");
        Thread.sleep(2000);
    }
     @After
     public void tearDown() throws Exception
     {}
}
```

Fig. 3. A selenium RC Junit test for patient registration page of hospital management system.

```
<mappings>
   <mapping>
      <phrase>state</phrase>
      <value>Patient Reg</value>
      <code>
      selenium=new DefaultSelenium("localhost",4444,"firefox/Applications/"
      +"Firefox.app/Contents/MacOS/firefox-bin","http://localhost:8089/");
      selenium.start();
      selenium.open("http://localhost:8089/Jkef/PatientRegistrationPage");
      </code>
   </mapping>
   <mapping>
      <phrase>username1</phrase>
      <value>u1</value>
      <range>non blank</range>
      <code>selenium.type("//input[@name='username']",u1);</code>
   </mapping>
   <mapping>
       <phrase>password1</phrase>
      <value>p1</value>
      <range>non blank</range>
      <code>selenium.type("//input[@name='password']",p1); </code>
   </mapping>
   <mapping>
      <phrase>Name</phrase>
      <value>n</value>
      <range> alphanumeric </range>
      <code>selenium.type("//input[@name='name']",n);</code>
   </mapping>
   <mapping>
      <phrase>Age</phrase>
      <value>ag</value>
      <range>integer less than 30</range>
      <code>selenium.type("//input[@age='age']",ag); </code>
   </mapping>
   <mapping>
      <phrase>register.clicked</phrase>
      <value>submit</value>
      <code>selenium.click("//input[@name='submit']");
      selenium.waitForPageToLoad("30000");
      </code>
   </mapping>
</mappings>
```

Fig. 4. Mapping between phrases used in model to selenium RC Junit code.

VI. Conclusions and Future Work

In this paper, we modeled our case study application using two formal specification languages, State chart and Z notation. We derived abstract test cases from these two formal specification languages and gave a mechanism for an equivalence between the two proposed frameworks. We then presented a mapping between phrases used in the statechart model to Selenium RC Junit test code which will help to translate an abstract test to concrete Selenium Junit test, which can be run directly on the system under test. The mapping xml file was currently created manually from the implementation of the case study of Hospital Management System. We are working on automatically mapping abstract test case to Selenium RC Junit concrete test cases for any given web application.

Appendix

Formal web business logic specification using Z notation.

$[PATIENT]$

$$RESPONSE ::= patientLoginSuccess \mid patientLoginFailure \mid patientRegistrationSuccess \mid patientRegistrationFailure$$

Hospital

$$loggedInPatients : \mathbb{P}\ PATIENT$$
$$registeredPatients : \mathbb{P}\ PATIENT$$
$$loggedInPatients \subseteq registeredPatients$$

InitialHospital

$$Hospital'$$
$$registeredPatients = \emptyset$$

PatientRegisterSuccess

$$\Delta Hospital$$
$$patient? : PATIENT$$
$$response! : RESPONSE$$
$$patient? \notin registeredPatients$$
$$registeredPatients' = registeredPatients \cup \{patient?\}$$
$$response! = patientRegistrationSuccess$$

PatientRegisterFailure

$$\Xi Hospital$$
$$patient? : PATIENT$$
$$response! : RESPONSE$$
$$patient? \in registeredPatients$$
$$registeredPatients' = registeredPatients$$
$$response! = patientRegistrationFailure$$

$$PatientRegister \mathrel{\widehat{=}} PatientRegisterSuccess \lor PatientRegisterFailure$$

PatientLoginSuccess

$$\Delta Hospital$$
$$patient? : PATIENT$$
$$response! : RESPONSE$$
$$patient? \notin loggedInPatients$$
$$loggedInPatients' = loggedInPatients \cup \{patient?\}$$
$$response! = patientLoginSuccess$$

$$
\begin{array}{|l}
\hline
PatientLoginFailure \\
\Xi Hospital \\
patient? : PATIENT \\
response! : RESPONSE \\
\hline
patient? \in loggedInPatients \lor patient? \notin \\
registeredPatients \\
loggedInPatients' = loggedInPatients \\
response! = patientLoginFailure \\
\hline
\end{array}
$$

$$PatientLogin \widehat{=} PatientLoginSuccess \lor PatientLoginFailure$$

REFERENCES

[1] K. Homma, S. Izumi, K. Takahashi, and A. Togashi, "Modeling and verification of web applications using formal approach."

[2] M. Han and C. Hofmeister, "Modeling and verification of adaptive navigation in web applications," in *Proc. ICWE '06*, July 11- 14, 2006, Palo Alto, California, USA, ACM.

[3] R. Filippo and T. Paolo, "Analysis and testing of web applications," in *Proc. the 23rd International Conference on Software Engineering*, 2001.

[4] N. Li and J. Offutt, "A test automation language for behavioral models," Technical Report, GMU-CS-TR-2013-7.

[5] B. Zhu, H. K. Miao, H. W. Zeng, and S. B. Chen, "Generating test case from functional requirements of web applications," in *Proc. Second International Symposium on Electronic Commerce and Security*, 2009.

[6] J. P. Gruer, V. Hilaire, A. Koukam, and P. Rovarini, "Heterogeneous formal specification based on Object-Z and statecharts: Semantics and verification," *The Journal of Systems and Software*, vol. 70, 2004.

Khusbu Bubna has been an MS student in International Institute of Information Technology, Bangalore since 2013. She received her B.Tech. degree in computer science and engineering from Institute of Engineering and Management, Kolkata, India in 2013. Her current research interests include formal verification and test generation of web applications.