ACT (Abstract to Concrete Tests) - A tool for generating Concrete test cases from Formal Specification of Web Applications

Khusbu Bubna
International Institute of Information Technology, Bangalore
khusbu.bubna@iiitb.org

Sujit Kumar Chakrabarti
International Institute of Information Technology, Bangalore
sujitkc@iiitb.ac.in

ABSTRACT

As web applications are becoming more and more ubiquitous, modeling and testing web applications correctly is becoming necessary. In this paper, we have used a formal specification language, State chart to model the navigation behaviour aspect of web applications. This paper presents the ACT (Abstract to Concrete Tests) tool, an approach of generating concrete executable Selenium RC JUnit test cases from a formal State chart specification model. The ACT tool can generate concrete Selenium RC JUnit test cases from abstract test cases by utilizing data shared across different interactions of the web application with the web server. Throughout the paper, a case study of Learning Management System is used to illustrate our approach.

1. INTRODUCTION

Nowdays, the use of web applications has grown to a huge extent. Web applications are used in online shopping, online banking, etc., so designing and testing web applications rigorously has become very crucial. Formal specification languages are used to clarify customer's requirements by removing ambiguity, inconsistency and incompleteness in the software requirements and design process. Thus, they are helpful in reducing the requirement errors in the early stages of the software development life cycle. Formal specifications are mostly used in critical systems where the cost of failure is catastrophic. Formal specifications can be used in several ways of design cycle, static verification and test generation being the most notable.

In this paper, we present the ACT (**A**bstract to **C**oncrete **T**ests) tool, which uses a novel approach of generating concrete test cases from abstract test cases by using data shared across different interactions of the web application with the web server. A formal specification language 'Statecharts' [1] was used to model the navigation behaviour aspect of web applications. Then abstract test cases were generated from the State chart model in our ACT tool. Abstract test cases cannot be directly executed on the implementation since they are defined using elements and objects of the State chart model, and must be first transformed to concrete test cases. So finally, concrete executable Selenium RC JUnit test cases are generated from these abstract test cases in our ACT tool which can be directly executed on the implementation. The remainder of the paper is organized as follows. Section 2 gives a brief review of related work, while Section 3 explains the proposed methodology used by our ACT tool. Section 4 illustrates the State chart specification for modeling the navigation behaviour of web applications. Section 5 illustrates the generation of test path from the State chart model. Section 6 illustrates how abstract test cases are generated using Symbolic execution and SMT solver. Section 7 illustrates the translation of abstract test cases to concrete test cases. Section 8 concludes the paper and Section 9 gives scope for future work.

2. RELATED WORK

A number of formal, informal and semi-formal models like automata [2], Statechart [3], UML and OCL [5], UML based web engineering, alloy, directed graph and control flow graphs, SDL, term rewriting systems, XML have been proposed in various studies [6] for modeling web applications. The authors in [5] have proposed UML class diagram and the authors in [3] have proposed statechart for modeling web navigation. A methodology for generation of concrete executable tests from abstract test cases using a test automation language, the Structured Test Automation Language (STAL) was proposed in [7]. The authors in [7] have proposed a mapping between identifiable elements in the model to JUnit executable java code. The author in [8] have presented an approach using domain specific languages to model the navigation aspect of the web application and have used a UI mapping XML file to generate concrete test cases for Selenium and Canoo web test tools. In [8], an approach that utilizes recorded user interaction data to construct a state machine model especially for testing AJAX functionality is presented. Input data is provided from the collected requests and test oracles have to be created manually. The generated test sequences are translated into the test case format of the Selenium test automation tool. The authors in [9] have used model checking to generate test cases for control flow and data flow coverage criteria. The authors in [10]

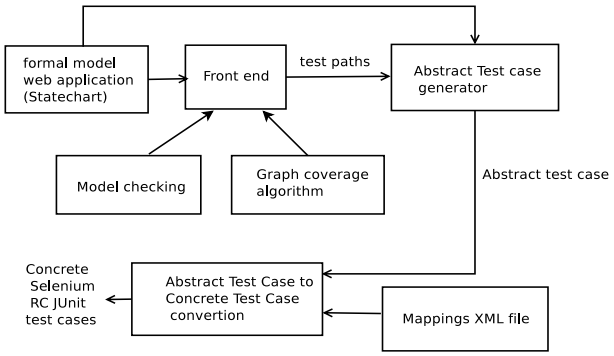

Figure 1: Proposed Test Generation Method used by our ACT tool

3. PROPOSED METHODOLOGY

Figure 1 gives an overview of the approach that the ACT (Abstract to Concrete test) tool uses for generating concrete Selenium RC JUnit test cases from the formal State chart web navigation model. The navigation behaviour of our case study web application is modeled using the formal specification language 'State chart'. The front end which generates test paths from the State chart model can be a test path generating algorithm like model checking or graph coverage algorithm. Abstract test cases are generated from the test paths with the help of the State chart specification. Then the abstract test cases generated are converted to Selenium RC JUnit concrete test cases using the Mappings XML file. Each of these steps are explained in elaborate details in the remaining sections.

4. A FORMAL MODEL FOR WEB NAVIGATION - STATE CHART

From among the various finite state based formal specification languages, we have used Statecharts [1] for modeling the navigation behaviour of web applications. Figure 2 shows the State chart specification of our case study, the Learning Management System (LMS) used within our institute. In the Statechart specification shown in Figure 2, *Inactive* state denotes that the web application has not yet started operating. The composite state *Active* denotes that the web application is in an operating state. Inside *Active* state, each web page was modelled as a separate state. When the user navigates from one webpage to another, there is a transition between the corresponding states. *LMS Login Page* denotes the Login page of Learning Management System. Here we have used the mathematical notations of sets and first order predicate logic constructs in guards and actions of the transitions in State chart model. In a web application, the value which is entered by the user in one interaction of the web application with the web server is often used back in another interaction of the web application with the web server. For example, as shown in the State chart model of Figure 2, admin can register a student in the LMS System by entering values in the username and password input fields in the *Add Student User Page*. A necessary requirement for a student to login in the *LMS Login Page* is that the student must be registered beforehand and so must have a valid student username and password. Thus there is a data flow between Add Student User Page page and LMS Login Page. The variables *valid_student_usernames* and *valid_student_passwords* are defined in the transition from state *Add Student User Page* to *Successfully Student Added Page* and are used in the transition from state *LMS Login Page* to *My Courses* web page. The variables *valid_student_usernames*, *valid_student_passwords*, *valid_teacher_usernames* and *valid_teacher_passwords* are shared between different interactions of the web application with the web server and across different users using the web application. When the web application has started operating, these variables are initialized to the null set. For example as shown in the State chart diagram of Figure 2, variables *valid_student_usernames*, *valid_student_passwords*, *valid_teacher_usernames* and *valid_teacher_passwords* are initialized to null set in the transition from state *Inactive* to state *Active*. On a successful registration of either a teacher or a student, the values of these variables are updated and when either a student or a teacher logs in the LMS Login page, the value of these variables are accessed to check if a student or teacher is already registered or not. Our ACT tool, as illustrated in the remaining sections can store and share data across different interactions of the web application with the web server and can generate Selenium RC JUnit concrete test cases utilizing this data.

5. TEST PATH GENERATION FROM STATE CHART MODEL

The front end step in the ACT tool which generates test paths can be any test path generating algorithm like graph coverage algorithms. But model checking which is a formal verification technique can also be used to generate test paths by formulating a temporal logic specification as a trap property to be verified. The straight forward way to represent a State chart as a transition system is to flatten its hierarchy. In our approach, the hierarchical State chart navigation model was first flattened and then transformed into an SMV program. The trap properties for navigation were written in CTL formulas and then the Symbolic Model Verifier (NuSMV) tool is executed which generates the counter examples. For example, the CTL Trap Specification Property for specifying that state SE-230_Software_Testing is reachable from the initial state is:

CTL Trap Specification Property:
! EF(state=SE-230_Software_Testing)

This trap property will generate a counter example which is our test path. For this counter example there will be two paths. One from LMS Login Page to My Courses Page to SE-230 Software Testing Page. And the second test path will be the loop from LMS Login Page to Successfully Student Added Page then to My Course Page to SE-230 Software Testing page. Here we will restrict the length of the loop in the counter example by using bounded model checking and set the length of the counter example to a fixed size. The CTL Trap properties for generating test paths are written for various requirements of the web application, like the top page of a web application should be reachable from all the pages of the web application. In addition, we also wrote CTL trap properties for node coverage criterion.

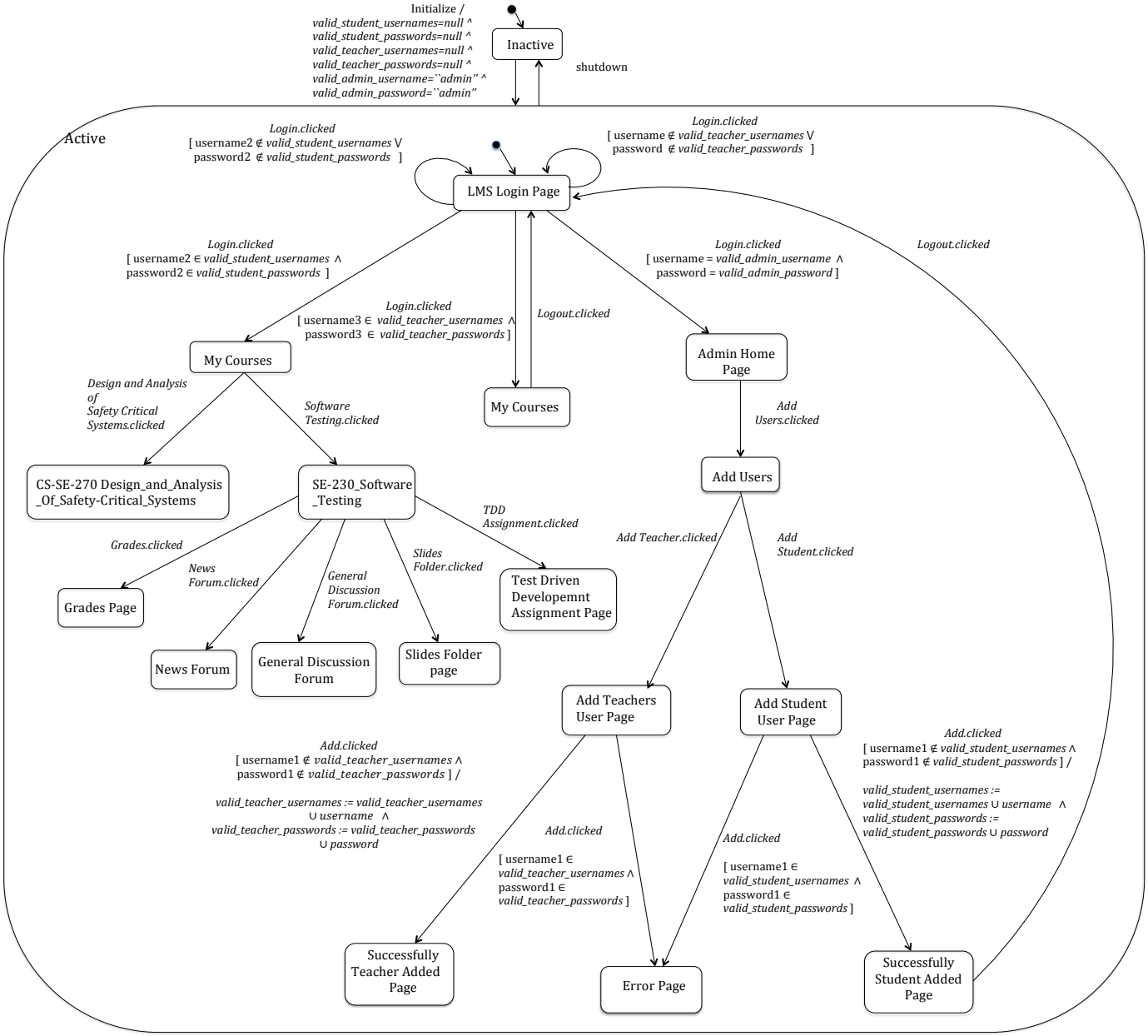

Figure 2: Formal Model showing Web Navigation of Learning Management System using UML State chart model.

Test Criterion	CTL Trap Property	Description
Node coverage	! EF (state=SE-230_Software_Testing)	A CTL Trap property for node coverage for state SE-230 Software Testing Page.
The top page is reachable from all the pages	! AG((state=SE-230_Software_Testing) → EF(state=LMS_Login_Page))	LMS Login Page is reachable from SE-230 Software Testing Page.
Every page is reachable from the top page	! AG ((state=LMS_Login_Page) → EF (state=SE-230_Software_Testing))	SE-230 Software Testing Page is reachable from LMS Login Page page.

Table 1: Some CTL Trap Temporal Logic Properties that were used for generating test paths from flattened State chart model of Learning Management System in NuSMV tool.

6. ABSTRACT TEST CASES GENERATION USING SYMBOLIC EXECUTION AND SMT SOLVER

After the test paths are generated from the State chart model, symbolic execution is carried out to generate input values to guide the execution along the test paths. The path predicate corresponding to the test path is computed. This, in turn, is given to an SMT solver to generate concrete input values. These values are used to generate the abstract test cases. The algorithm used to generate the abstract test cases using symbolic execution is shown in the block diagram in Figure 3.

6.1 Path Predicate Formation & Computation of Concrete Values

This section illustrates the method of formation of path predicates and the generation of concrete values using the example of the counter example generated from the CTL Trap Property *!EF(state=SE-230_Software_Testing)*. The variables *valid_student_usernames* and *valid_student_passwords* in the State chart model in Figure 2 are internal program variables and are internally maintained as sets. These variables are initialized to null in the transition from state *Inactive* to composite state *Active* as shown in Figure 2. The path predicate formed for the transition from *Add Student User Page* to Successfully Student Added Page is $(username1 \neq \text{""}) \wedge (password1 \neq \text{""}) \wedge (username1 \notin \phi) \wedge (password1 \notin \phi)$.

Sets *valid_student_usernames* and *valid_student_passwords* are consequently updated to *username1* and *password1*. So the path predicate for the transition from state *LMS Login Page* to *My Courses* is $((username1 \neq \text{""} \wedge (password1 \neq \text{""}) \wedge (username1 \notin \phi) \wedge (password1 \notin \phi) \wedge (username2 \in username1) \wedge (password2 \in password1))$
So the final path predicate for the counter example *! EF(state=SE-230_Software_Testing)* is: $((username1 \neq \text{""} \wedge (password1 \neq \text{""}) \wedge (username1 \notin \phi) \wedge (password1 \notin \phi) \wedge (username2 \in username1) \wedge (password2 \in password1))$
Here *username1* and *password1* are the symbolic variables corresponding to the Add_Student_User page, and *username2* and password2 are the symbolic variables corresponding to the LMS Login page. An SMT solver like CVC3 [11] would determine if the path predicate is satisfiable, and if yes, then it will generate satisfying values for the symbolic variables occurring in the formula and hence the concrete input values would be generated. Suppose, if a counter example is generated for the path from state Inactive to state *Error Page* through states LMS Login, *Admin Home Page, Add Users* and *Add Student User Page*, then the path predicate formed will be: $(username1 \in \phi) \wedge (password1 \in \phi)$ which will be a contradiction i.e. unsatisfiable predicate. Hence, this path will be rejected as infeasible.

6.2 Abstract Test Cases Generation

Finally the abstract test cases are generated by plugging in concrete input values which are computed in the symbolic execution stage at appropriate points in the counter examples. The phases in the counter examples which have

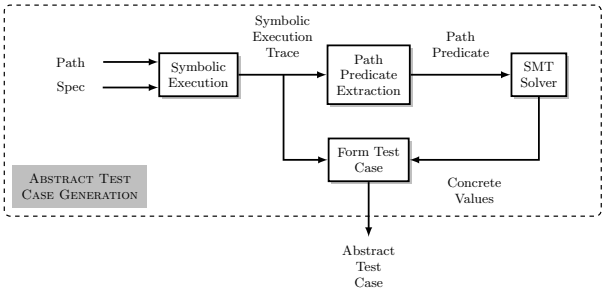

Figure 3: Abstract test case generation using Symbolic execution

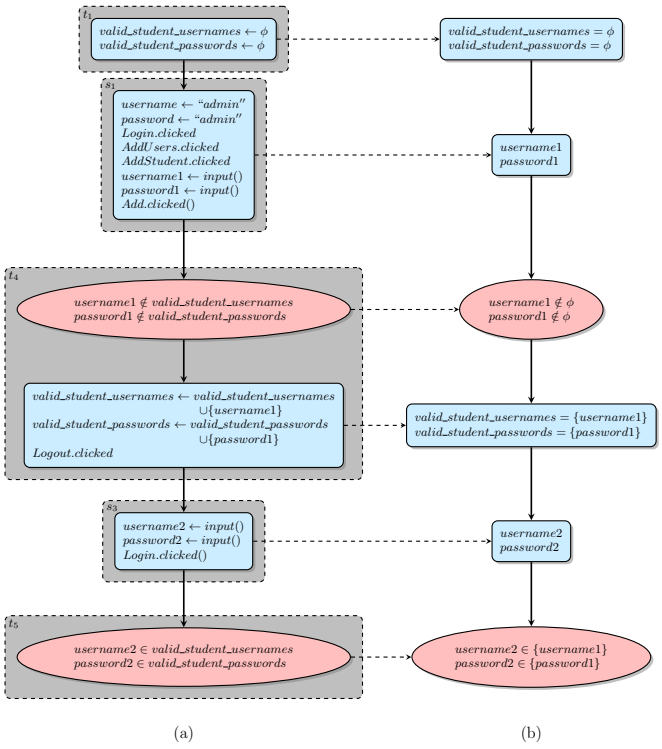

Figure 4: Symbolic Execution: (a) Control flow path; (b) Symbolic Execution Trace

corresponding GUI elements in system implementation are extracted in sequence and their user input fields are replaced by concrete values computed in symbolic execution stage.

7. TRANSLATION OF ABSTRACT TEST CASES TO CONCRETE TEST CASES

The last step is deriving concrete executable test cases from these abstract test cases so these concrete test cases can communicate directly with the system under test. The abstract test cases generated from the State chart model are on the same level of abstraction as the model since the State chart model they are generated from contains only partial information of the implementation under test. In Figure 5, even through the input values are concrete, the test case is

username="admin"
password="admin"
Login.clicked
Add Users.clicked
Add Student.clicked
username1="MS2013009"
password1="abc"
Add.clicked
Logout.clicked
username2="MS2013009"
password2="abc"
Login.clicked
Software Testing.clicked

Figure 5: An example of an abstract test case generated for the counter example generated from the CTL Trap Temporal Logic Property *!EF(state=SE-230_Software_Testing)*

still abstract because firstly variables like username1, password1, etc. needs to be mapped to the implementation entities . And, secondly many of the inputs that may be needed may not be there in the original specification and are only available after the exact implementation is finished. For generating concrete test cases from the abstract test cases generated in the previous step, the ACT tool uses a mapping between phrases used in the State chart specification model to Selenium Remote Control JUnit test code which helps to translate an abstract test to concrete Selenium RC JUnit test. Figure 8 shows a part of XML file that was created to give a mapping between the phrases in the State chart model of Figure 2 to Selenium RC JUnit java code for our case study of Learning Management System. The mappings XML file is created manually. Since, the State chart model contained only partial information of the final implementation. So, now the additional information that were abstracted out from the State chart model has to be added in the Mappings XML file in order to generate the concrete Selenium RC Junit test cases. From the source code implementation of the case study of HMS, we found that in the Add Student User page there were additional input fields like Name and Department which were abstracted out in the State chart specification of Figure 2. So these fields were also included in the Mappings XML file. Using the generated abstract test cases from the previous step and the mappings XML file created manually and fed as input to the ACT tool, a set of concrete Selenium RC Junit test cases is generated in this step. Figure 5 shows one of the concrete Selenium JUnit test case generated from our ACT tool for the case study of Learning Management System.

8. RESULTS

For two other web based enterprise applications developed within our institute, a Hospital Management System (HMS) and a Student Information System (SIS), we modeled our web applications using the proposed method.

Table 2 shows the lines of code (LOC) of the two systems, the number of nodes in the flattened State chart (N_{flat}), the number of mappings and the number of abstract test cases(or counter examples) used for the various test coverage criterion. A total of 14 abstract test cases were used for HMS

```
<mappings>
  <mapping>
    <phrase>state</phrase>
    <value>Add_Student_User</value>
    <code>
      selenium=new DefaultSelenium("localhost",4444,"firefox/"
                                   +"Applications/Fire-
fox.app/Contents/MacOS/"                              +"firefox-
bin","https://lms.iiitb.ac.in/moodle");"
      selenium.start();
      selenium.open("/course/view.php?id=19");
    </code>
  </mapping>
  <mapping>
    <phrase>username1</phrase>
    <value>u1</value>
    <range>non blank</range>
    <code>selenium.type ("css=input[id='username',]",u1);</code>
  </mapping>
  <mapping>
    <phrase>Name</phrase>
    <value>n</value>
    <code> selenium.type("css=input[id='name']",n); </code>
  </mapping>
  <mapping>
    <phrase>Department</phrase>
    <value>dp</value>
    <code> selenium.type("css=input[id='department']",dp);</code>
  </mapping>
</mappings>
```

Figure 6: A part of Mappings XML file showing mappings between phrases used in State chart specification model of Figure 2 to their corresponding Selenium RC JUnit code

system and a total of 53 abstract test cases were used for SIS System for various test criterion. Currently, we have modelled the two case studies of HMS System and SIS System but we are currently running our ACT tool and trying to calculate the number of concrete test cases that would be generated.

9. CONCLUSION

Test cases generated from formal specifications are often at an abstract level. They cannot be executed directly using a test automation tool, e.g. Selenium. The existing methods have tried to deal with this problem using a mapping approach. However, this approach is fundamentally limited in its capability to translate abstract test cases into concrete test case due to the presence of data flow relationship between atomic interactions between the (test-)client and the server. In this paper, we have presented a precise characterisation of this problem. Further, we have provided a test generation methodology which uses symbolic execution to resolve the data-flow between atomic interaction. The resulting abstract test cases are concrete enough so that the mapping method is effective in completely translating them to concrete test cases that can be directly executed on Selenium test runner. Through some of the steps like creation of Mappings XML file, construction of State chart, etc are manual but there is fairly a reasonable automation in the other steps in the ACT tool. Therefore, our approach is able to achieve a reasonable amount of end to end automation of the test generation process from formal specification.

Web Applications	N_{flat}	LOC	No. of Mappings	No. of abstract tests (counter examples)	
Hospital Management System (HMS)	6	133	20	Node coverage	6
				Top page is reachable from all the pages	4
				Every page is reachable from the top page	4
				Total	**14**
Student Information System (SIS)	19	15,770	56	Node coverage	19
				Top page is reachable from all the pages	17
				Every page is reachable from the top page	17
				Total	**53**

Table 2: Results of test generation on two other case studies Hospital Management System and Student Information System

10. FUTURE WORK

In our future work, we will try to address the following issues:

- Currently we are running our ACT tool on our LMS system and we are currently trying to find out the number of concrete test cases that would be generated for various test criteria. So in our future work, we will try to find out the relation between the number of concrete test cases and the number of abstract test cases and we will also try to see how the number of concrete test case and the number of abstract test cases varies with the size of the State chart model.
- In our future work, we will generalise this approach to apply other forms of formal specification and coverage criteria. In addition, we intend to apply other forms of front-end techniques like graph coverage criteria in generating test paths instead of the model checking approach for generation of test paths .

11. REFERENCES

[1] D. Harel,"Statecharts: A Visual formalism for complex systems", Science of Computer Programming, Volume 8, Issue 3, June 1, 1987, Pages-231-274.

[2] K. Homma, S. Izumi, K. Takahashi and A. Togashi,"Modeling and Verification of Web Applications Using Formal Approach". IEICE Tech. Report, 2009.

[3] M. Han,C. Hofmeister,"Modeling and Verification of Adaptive Navigation in Web Applications", Proc. of 6th Intl. Conf. on Web Engg., pp. 329-336.

[4] F. Ricca, and P. Tonella, "Analysis and Testing of Web Applications", Proc. of the 23rd Intl. Conf. on Software Engineering, pp. 25-34, 2001.

[5] M. H. Alalfi, J. R.Cordy, T. R.Dean,"Modeling methods for Web Application Verification and Testing: State of the art", Software Testing, Verification and Relaibility, Vol. 19, Issue 4, pp. 265-296.

[6] N. Li, J. Offutt, "A Test Automation Language for Behavioral Models", Technical Report, GMU-CS-TR-2013-7.

[7] A.-M. Torsel, "A Testing Tool for Web Applications Using a Domain-Specific Modelling Language and the NuSMV Model Checker", 6th Intl. Conf. on Software Testing, Verification and Validation, pp. 383-390.

[8] A. Marcetto, P. Tonella, and F. Ricca, "State-based Testing of AJAX Web Applications", in Proc. of 2008 Intl. Conf. on Software Testing, Verification, and Validation, pp. 121-130.

[9] H. S. Hong, I. Lee, O. Sokolsky, "Automatic Test Generation from Statecharts Using Model Checking", in Proc. of FATES'01, Workshop on Formal Approaches to Testing of Software, Vol. NS-01-4 of BRICS Notes Series.

[10] S. Gnesi, D. Latella, and M. Massink, "Formal Test-Case Generation for UML Statecharts," Proc. Ninth IEEE International Conf. Eng. of Complex Computer Systems, P. Bellini, S. Bohner, and B. Steffen, eds., Apr. 2004.

[11] CVC3 SMT solver: "http://www.cs.nyu.edu/acsys/cvc3/"

```
public void test1()
{
  selenium.start();
  selenium.open("ROOT/login/index.php");
  selenium.type("css=input[id='username']",
   "admin");
  selenium.type("css=input[id='password']",
   "admin");
  selenium.click("xpath=//a[contains(@href,
   'ROOT/myadmin/')]");
  selenium.click("xpath=//a[contains(@href,
   'ROOT/user/index.php?id=27')]");
  selenium.click("xpath=//a[contains(@href,
   'ROOT'
  selenium.type("css=input[id='username1']",
   "Joe");
  selenium.type("css=input[id='password1']",
   "abc");
  selenium.click("//input[@id='loginbtn']");
  selenium.click("xpath=//a[contains(@href,
   'ROOT/login/index.php')]");
  selenium.type("css=input[id='username2']",
   "Joe");
  selenium.type("css=input[id='password2']",
   "abc");
  selenium.click("xpath=//a[contains(@href,
   'ROOT/my/')]");
  selenium.click("xpath=//a[contains(@href,
   'ROOT/course/view.php?id=19')]");
}
```

Figure 7: A Selenium RC JUnit concrete test case generated from our ACT tool for the case study of Learning Management System for the test path generated from the CTL trap property *!EF(state= SE-230_Software_Testing)*