End-to-End Specification Based Test Generation for Web-Applications

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

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

ABSTRACT

Using formal specifications to generate test cases presents great potential for automation in testing and enhancing the quality of test cases. However, an important challenge in this direction is that specifications are at a more abstract level than the implementation, with many important implementation level details missing. But to generate executable test cases, these implementation details must be included at some stage. Though, there has been a lot of work in test generation from specification, all existing methods suffer from this problem: either the test cases are not executable, or the process involves a non-trivial manual step of translating the abstract test cases to concrete test cases.

In this work, we present an approach of specification based test generation for web applications, called ACT, that overcomes the above challenge: test generation is completely automated and the test cases are fully executable on a test execution framework (e.g. Selenium RC). Further, our methodology allows generation of multiple sets of concrete test cases from the the same formal specification. This makes it possible to use the same abstract specification to generate test cases for a number of versions of the system. Using ACT, we generated concrete Selenium RC Junit test cases for two web applications, Hospital Management System (LOC 133) and Student Information System. The concrete test cases obtained were executed on the implementations of these systems.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing tools, symbolic execution; H.3.5 [**Online Information Services**]: Web-based services

General Terms

Algorithms, measurement, experimentation

Keywords

Test generation, embedded systems, symbolic execution

1. INTRODUCTION

Due to the pervasive use of web applications, building in rigour in designing and testing web applications [1] (web apps) has assumed greater importance in recent years. Formal specification languages are used to elicit customer requirements by helping remove ambiguity, inconsistency and incompleteness in the software requirements and design process. Formal methods have been widely used for verification and test generation for life-critical software systems, e.g. automotive and aerospace. However, their adoption in web apps has been limited, primarily due to cost consideration.

Focusing on the problem of specification based test generation for web apps, in our observation, there are two main aspects of this problem: *cross-interaction data dependency* (or just *data dependency*) and *mapping*. To understand the data dependency problem, note that a test case for a web app is usually a sequence of atomic interactions (consisting of inputs to and outputs from the system under test). In most realistic cases, values involved in one interaction are related to other interactions. For example, for a successful login, it is necessary to enter a user name which is already registered (through an earlier interaction). Similarly, the password used must correspond to the user name entered. Such inter-relations between data values from various atomic interaction are a tricky aspect of specification based test generation. Any test generation method using an isolated fragment of the formal specification at a time will fail to capture these cross-interaction data dependencies.

The second problem is called the *mapping problem*. (Formal) Specification derives its value in its succinctness and abstractness. Unnecessary implementation details are left out, making it more understandable to human readers and amenable to automated analysis. However, due to the same reason, test cases generated from specification also are *abstract*. Some of the details necessary to execute these test cases are missing, as they are not there in the source itself (the specification). Therefore, to execute such test cases, they must be *concretised*, i.e. all the necessary implementation details must be re-introduced. Little automation has been achieved in this direction in the literature. Therefore, this translation of abstract test cases to concrete (executable) test cases is done largely manually. This significantly diminishes the benefits derived from automated test generation from specification.

In this paper, we present ACT (i.e. **A**bstract to **C**oncrete **T**est), a specification based test generation method that automatically generates concrete test cases from abstract formal specification (a variant of Statecharts to model the navigation behaviour of web applications). ACT solves the data dependency problem using symbolic execution, a technique that has been successfully used for test case generation in a wide variety of settings. To solve the mapping problem, we use a novel mapping approach that automatically translates abstract test cases into concrete ones. By solving these two major hurdles, ACT completely automates the process of test generation from formal specification of web apps.

In our experiments, test cases for two web app were derived from the Statechart model. Abstract test cases derived from the Statechart model were converted to Selenium RC JUnit concrete test cases which were then successfully executed on the system implementation.

The remainder of the paper is organized as follows. Section 2 gives the background and motivation, Section 3 gives a brief review of related work while Section 4 explains the proposed methodology. Section 5 illustrates the generation of test path from Statechart 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 gives the results and Section 9 concludes the paper and gives the scope for future work.

2. MOTIVATION

Several studies [14, 5, 6] have used automation to generate concrete test cases from abstract test cases for testing web applications. The work in [6] uses Abstract State Machines (ASMs) to model web applications, and abstract test sequences are generated from the ASM model. In their approach, the abstract test case to concrete test case translator scans the abstract test sequence in order to extract the values of the event variable and concrete Sahi [7] scripts are generated according to these values. A template is used to describe the rules which guides the translation process. For example, the Sahi transformation rule for a click event is

SUBMIT(name) ::=<<_click(_submit("name"));>>.

Figure 1 shows a part of an abstract test sequence generated from ASM model, and Figure 2 shows the corresponding Sahi scripts generated using the abstract to concrete test case translator approach in [6].

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

Figure 1: An Abstract Test Sequence Example

However, the approach in [6] does not generate concrete test cases from abstract test cases when there is a data flow between different interactions of the web application with the web server. 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, Figure 3 shows an example of data flow across different interactions of an online Hospital Management system. As shown in Figure 3, a user registers in the *Patient_Reg* page by entering values in the username and password input fields. A necessary requirement for the user to login into the *Patient_Login* page is that the user must be registered beforehand. There is a data flow between the *Patient_Reg* and *Patient_Login* webpage. A **definition** is a location where a value for a variable is stored in memory (assignment, user input, etc.). A **use** is a location where a variable's value is accessed. Here, the variables *registered_Usernames* and *registered_Passwords* are defined in the transition from state *Patient_Reg* to *Patient_Login* and are used in the transition from state *Patient_Login* state to *Patient_Dashboard* state. So, our ACT (Abstract to Concrete Tests) tool needs to store the concrete test input values which it generates for the username and password fields in the *Patient_Reg* webpage, and then use these values when generating concrete test inputs later in the *Patient_Login* page. For this purpose, the automation step of conversion of abstract test case to concrete test case in our approach stores and shares data across different interactions of the web application with the web server and generates Selenium RC JUnit concrete test cases utilizing this data. For instance, if the concrete test input values

```
_navigateTo("index.php");
_setValue(_textbox("username"),"admin");
_setValue(_textbox("password"),"admin");
_click(_submit("submit"));
_assertEqual("main.php",top.location.href);
```

Figure 2: The corespnding Concrete Sahi Script of Figure 1

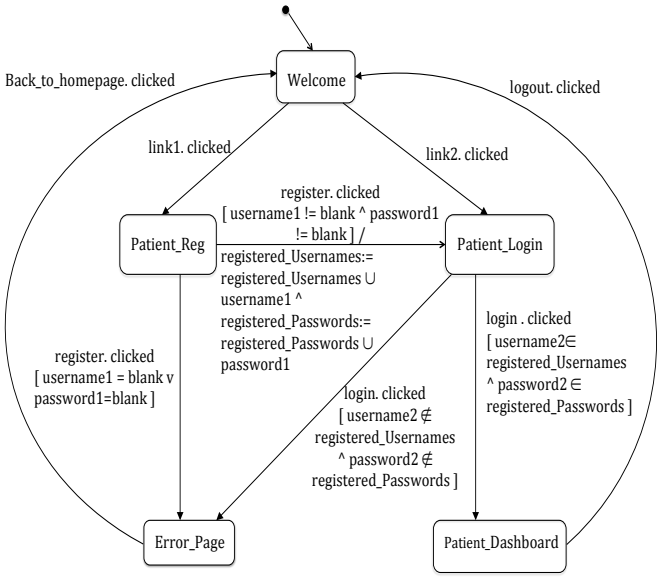

Figure 3: An example of data flow in Hospital Management System

generated for the username and password fields of the Patient Registration page (*Patient_Reg* in Figure 3) are "Joe" and "abc", then "Joe" and "abc" would be entered as test inputs in the Patient Login page (*Patient_Login* in Figure 3) in the case of Hospital Management System.

3. RELATED WORK

A number of formal, informal and semi-formal models like automata [8], Statechart [9], UML and OCL [11, 20], UML based web engineering, alloy, directed graph and control flow graphs, SDL, term rewriting systems, XML [23] have been proposed in various studies [12] for modeling web applications. The authors in [11, 20] have proposed UML class diagram and the authors in [9] 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 [14]. The authors in [14] have proposed a mapping between identifiable elements in the model to JUnit executable Java code. The author in [5] have presented an approach using domain specific language 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. However none of these approaches generate concrete test cases from abstract test cases by sharing and utilizing data across different interactions with the web application. In our approach, after the abstract test cases are derived from the Statechart model, we could generate concrete Selenium RC JUnit test cases by sharing data across different interactions with the web application. [15] used finite state machines for web application testing. In [16], 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 [21] have used model checking to generate test cases for control flow and data flow coverage criteria.

4. PROPOSED METHODOLOGY

4.1 Formal Web Navigation Model: Statechart

We have used Statecharts [10] for modelling the navigation behaviour of web applications.

Figure 4.1 shows the Statechart specification of our case study, an online Hospital Management System (HMS). The set of states in UML state diagrams represents both the *basic* states and the *composite* states which contain other states as sub-states. In the Statechart specification shown in Figure 4.1, *inactive* state denotes that the web application has not yet started operating. The composite state *active* is composed of five sub-states and denotes that the web application is in an operating state. The transition *initialize* takes the web application from *inactive* to *active* state and the variables *registered_Users* and *logged_Users* are initially set to null sets. The transition *shutdown* from *active* to *inactive* state denotes that the web application has stopped operating. Inside *active* state, each web page was modeled as a separate state. When the user navigates from one webpage to another, there is a transition between the corresponding states which is labeled by the tuple <*event*>[<*guard*>]/<*action*>, where only the <*event*> is mandatory. Here we have used mathematical notations of sets and first order predicate logic constructs in guards and actions of the transitions in Statechart model. *Patient_Reg* denotes the Patient Registration page of the Hospital Management System. *username1* and *password1* denotes the username andpassword that the user enters in the Patient Registration Page. *Patient_Login* denotes the Patient Login page, and *username2* and *password2* denotes the username and password which the user enters in the Patient Login page of the Hospital Management System.

In the given Statechart model of HMS, the variable *registered_Users* is 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, *registered_Users* is initialized to the null set. On a successful registration, its value is updated and while logging back, its value is accessed to check if the patient is already registered or not.

4.2 System Architecture

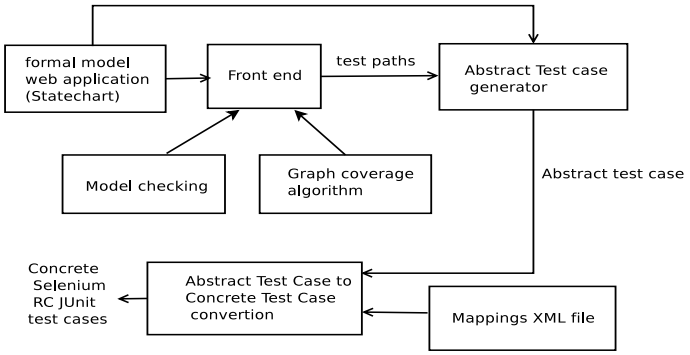

Figure 6: Proposed Test Generation Method from Statechart web navigation model

Figure 6 gives an overview of the approach that our ACT (Abstract to Concrete Tests) tool uses for generating concrete Selenium RC JUnit test cases from the formal Statechart web navigation model. The navigation behavior of our case study web application is modeled using the formal specification language 'Statechart'. The front end which generates test paths from the Statechart 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 Statechart specification. Then the abstract test cases generated are converted to Selenium RC JUnit concrete test cases using the Mappings XML file. We explain each of these steps in detail in sections V, VI and VII respectively.

5. TEST PATH GENERATION

There are several approaches proposed to generate test cases from Statechart model [18, 19]. We use model checking [3] as a front end to generate test paths from the Statechart model. Model checking is a formal verification technique

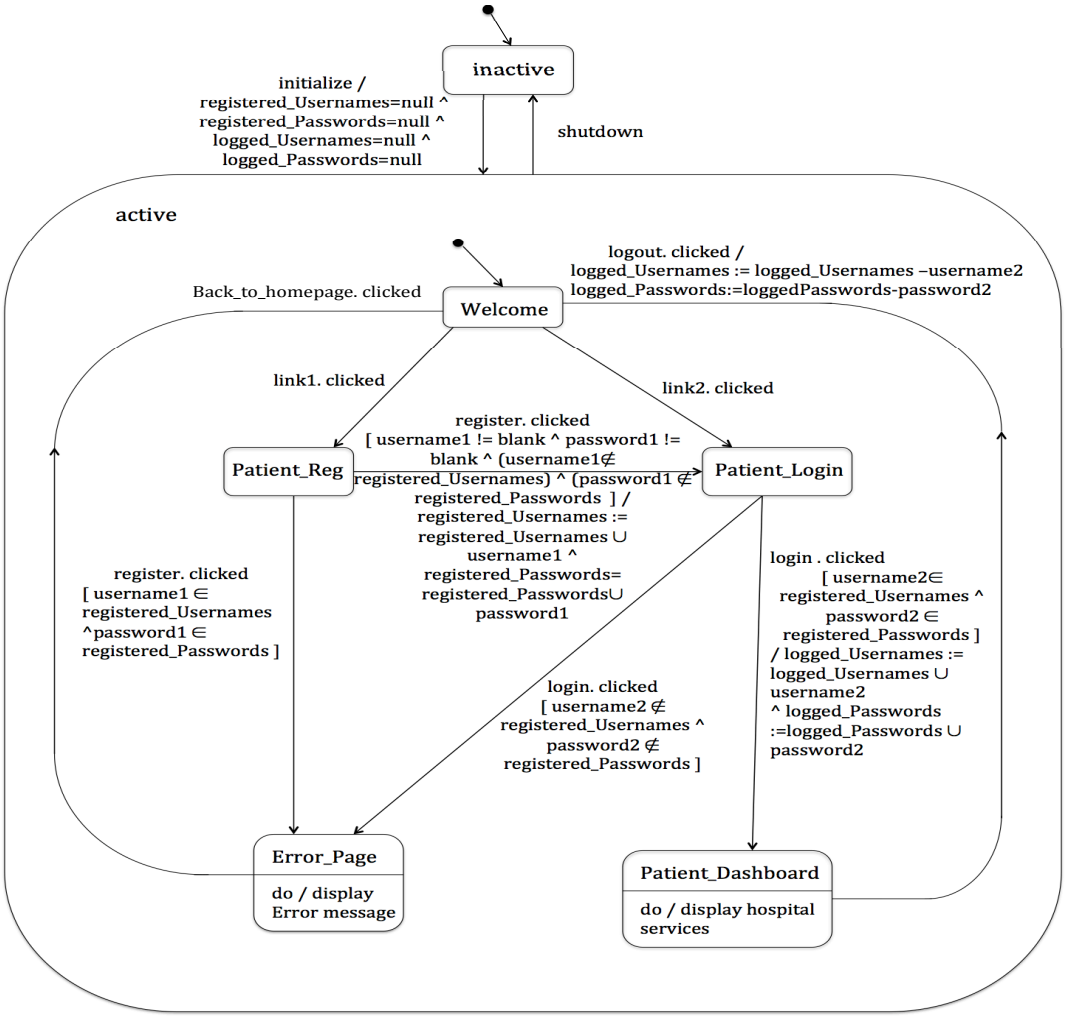

Figure 4: Model of Web Navigation of Hospital Management System using UML Statechart model.

which is used for determing 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. The straightforward way to represent a statechart as a transition system is to flatten its hierarchy [26]. Figure 5 shows the flattened statechart specification of Figure 4.1. In our approach, the hierarchical Statechart navigation model was first flattened and then transformed into an SMV program. The trap properties for navigation are written in CTL formulas and then the Symbolic Model Verifier (NuSMV) tool [22] is executed which generates the counter examples.

Test paths are 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. In this way, we get all the test paths from the generated counter examples. In the Statechart model of Figure 5, the state *active-Patient_Dashboard* should be reachable from the initial state in the Statechart. So, the CTL Specification property for specifying that *active-Patient_Dashboard* is reachable from the initial state is written as *EF(state=active-Patient_Dashboard)* which was negated to generate trap property as specified below.

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

This trap property will generate a counter example which is our test path. Figure 7 shows the execution trace for the counter example generated from NuSMV for the above CTL specification. 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. Table I shows some of the various CTL trap properties which we used to generate the test paths.

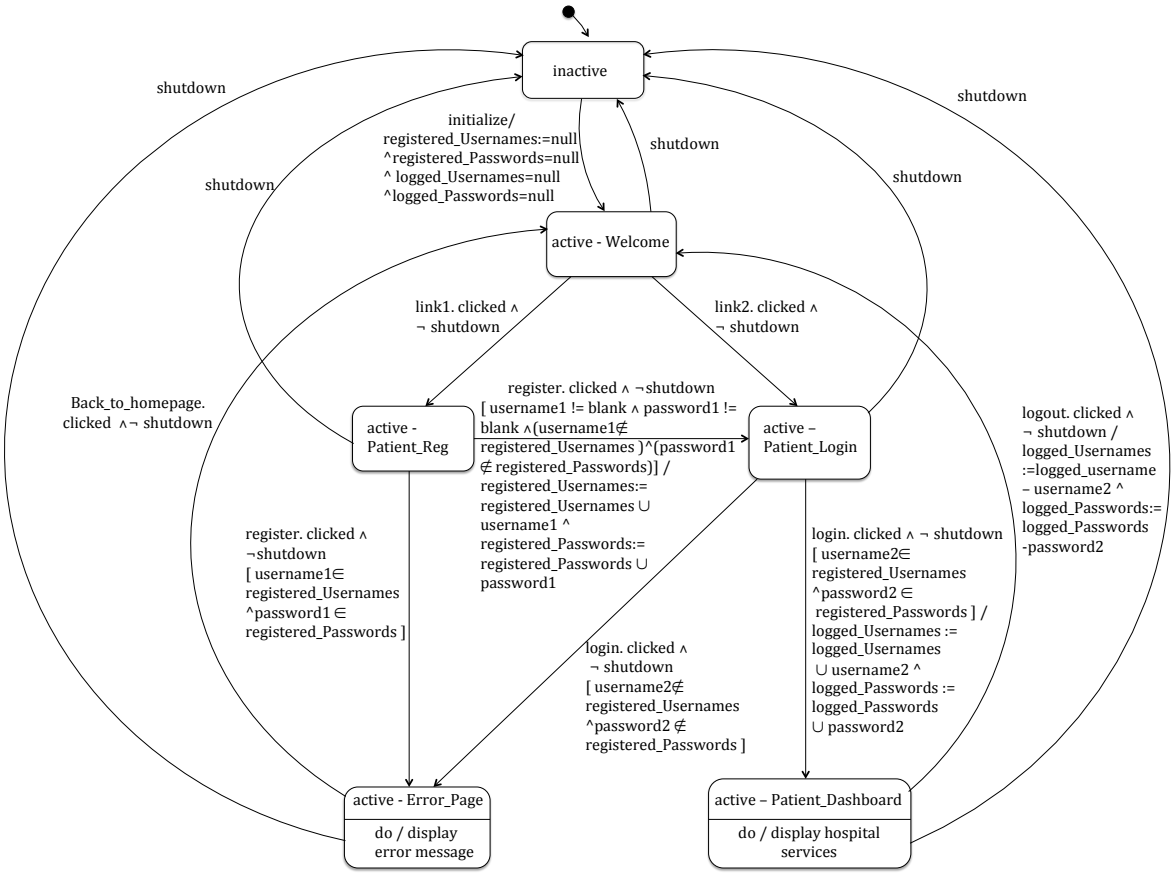

Figure 5: Flattened statechart specification of Hospital Management System given in Figure 4.1.

6. ABSTRACT TEST CASE GENERATION

The front-end of ACT (**A**bstract to **C**oncrete **T**ests) tool generates sytactic paths through the statechart. To generate concrete test cases from these paths, input values have to be computed which will lead the execution through the desired path. For example, consider the Statechart in Figure 9 which is a simplified version of the Statechart shown in Figure 5. Suppose that the front-end generates a path: $t_1t_4t_5$. To traverse t_2, the predicate in its guard, $u.text \notin RU$, should be satisfied. Since, $RU = \phi$ from the action of t_1, the above predicate is vacuously true. Thereafter, to traverse t_5, the predicate in its guard, $u.text \in RU$ should be satisfied. Since RU has only one value (the value assigned to $u.text$ on t_1), on t_5, $u.text$ must be assigned the same value to satisfy the above guard.

The above problem of generating input values to guide the execution along a particular path is well-known in test case generation. One of the most successful methods of solving this problem uses *symbolic execution* [13, 24]. The algorithm used by ACT to generate the abstract test cases using symbolic execution is shown in the block diagram in Figure 8. ACT carries out symbolic execution of the statechart specification along the given path. The *path predicate* corresponding to this 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.

We now explain the process of abstract test case generation using symbolic execution in further detail.

6.1 Computation of Concrete Values

The control flow path corresponding to the path $t_1t_4t_5$ is shown in figure 10(a). The rectangular blocks are basic blocks and the ellipses are decision blocks. The surrounding grey boxes represent the corresponding elements in the statechart. On executing this path symbolically, we get the symbolic execution trace shown in figure 10(b). The block by block mapping from the control flow path to the symbolic execution trace is shown using dashed arrows. The path predicate for a control flow path is derived by taking the conjunction of the predicates in symbolic execution trace blocks corresponding to its decision blocks (shown in ellipses). Therefore, for the path $t_1t_4t_5$, the path predicate is:

$$(X_1 \notin \phi) \wedge (X_2 \in \{X_1\})$$

where X_1 is the symbolic variable assigned to $s_1.u.text$ and X_2 is the symbolic variable assigned to $s_3.u.text$. An SMT solver [25] would determine if the path predicate is satisfiable, and if yes, then it will generate satisfying values for the symbolic variables occuring in the formula. For example, a value $X_1 = X_2 =$ `"a"` would satisfy the above path predi-

```
–specification ! (EF state=active-Patient_Dashboard) is false
–as demonstrated by the following execution sequence
Trace Description : CTL Counterexample
Trace Type : Counter example
  -> State : 6.1 <-
     state=inactive
     link.clicked=none
     initz=no
     back_to_homepage.clicked=no
     logout.clicked=no
     username1_password1=not_defined
     username2_password2=not_defined
     register.clicked=no
     login.clicked=no
  -> State : 6.2 <-
     initz=yes
  -> State : 6.3 <-
     state=active-Welcome
  -> State : 6.4 <-
     link.clicked=1
  -> State : 6.5 <-
     state=active-Patient_Reg
  -> State : 6.6 <-
     username1=nonblank_not_belongs_to_registered_Usernames
     password1=nonblank_not_belongs_to_registered_Passwords
     register.clicked=yes
  -> State : 6.7 <-
     state=active-Patient_Login
  -> State : 6.8 <-
     username2=belongs_to_registered_Usernames
     password2=belongs_to_registered_Passwords
     login.clicked=yes
  -> State : 6.9 <-
     state=active-Patient_Dashboard
```

Figure 7: Counter Example generated from NuSMV for the CTL specification !EF(state=active-Patient_Dashboard)

cate. Therefore, by entering the value `"a"` in both $s_1.u.text$ and $s_3.u.text$, we can lead the execution through the path $t_1t_4t_5$.

As another example, the syntactic path t_1t_2 would give a path predicate as follows:

$$(X_1 \in \phi)$$

which is a contradiction (i.e. unsatisfiable predicate). Hence, this path would be rejected by ACT as infeasible.

6.2 Plugging in Concrete Values

The final step of abstract test case generation is to plug in the concrete input values computed in the symbolic execution stage at appropriate points of the control flow path.

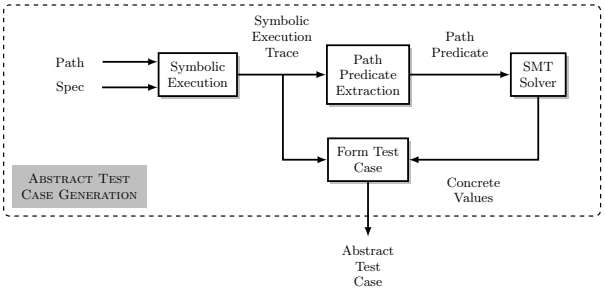

Figure 8: Abstract test case generation using Symbolic execution

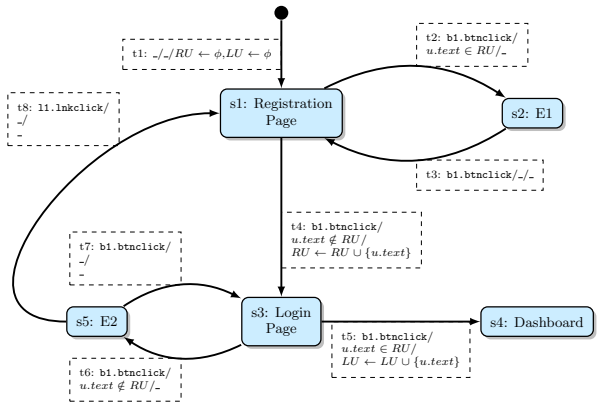

Figure 9: Statechart Specification *Registration* and *Login*

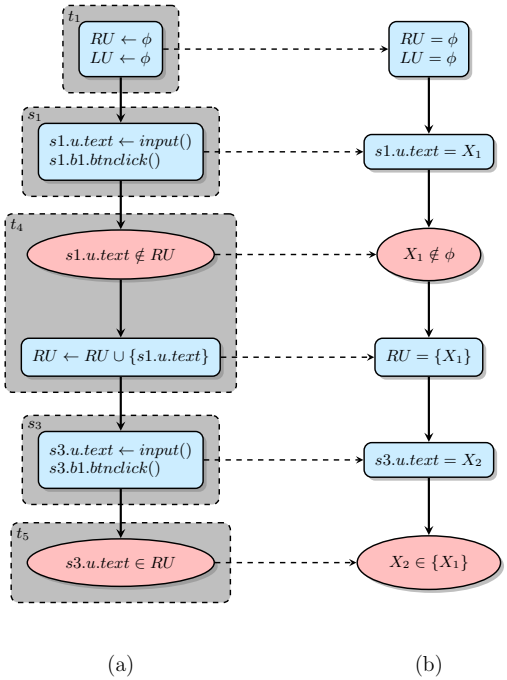

Figure 10: Symbolic Execution: (a) Control flow path; (b) Symbolic Execution Trace

The actions in all the basic blocks which cause definition of the input variables of the control flow path are taken in sequence to create the abstract test case. The *input*() commands are replaced by concrete values computed from the symbolic execution stage. This gives the complete abstract test case. This step would generate an abstract test case for the path $t_1t_4t_5$ as follows:

```
s1.u.text = "a"
s1.b1.btnclick()
s3.u.text = "a"
s3.b1.btnclick()
```

Note that even though the above inputs are concrete, the test case itself is still abstract. Firstly, the variables `s1.u.text` etc. need to be mapped to the implementation entities, e.g. *text box named "user name" in the page "Registration Page"*. Secondly, many of the inputs that may be needed in the various pages may not be there in the specification. This problem is solved using a mapping technique explained in the next section.

7. GENERATING CONCRETE TEST CASES

The abstract test cases generated from the Statechart model are on the same level of abstraction as the model. The generated test cases are abstract because the Statechart model they are generated from contained only partial information of the implementation under test. These abstract test cases cannot be directly executed on the implementation. An executable test suite needs to be derived from the abstract test suite which can communicate directly with the system under test. This is achieved by mapping the abstract test cases to concrete executable test cases. Selenium RC [27] is a browser automation testing tool that is used to write automated web application UI tests. In our approach, we have used a mapping between phrases used in the Statechart specification model to Selenium Remote Control JUnit test code which helps to translate an abstract test to concrete Selenium RC JUnit test. Figure 11 shows a part of XML file which gives a mapping between the phrases used in the Statechart model to Selenium RC JUnit java code for the case study of Hospital Management System. For example, the phrase for state *active-Patient_Reg* used in the Statechart model in Figure 5 is mapped to the Selenium RC JUnit code `selenium.open("http://localhost:8089/Jkek/PatientRegistrationPage")`; as shown in the mappings XML file in Figure 11. The mapping XML file contains the phrases used in the Statechart model denoted by <phrase> tag, the value of the phrase denoted by <value> tag, the webpage in which the phrase occurs denoted by <webpage> tag, constraints on the values that the phrase can take denoted by <range> tag and the corresponding Selenium RC JUnit test code denoted by <code> tag. These mappings are created manually.

At this stage, it is possible to add information for inputs that were abstracted out from the Statechart model. From the source code implementation of the case study of Hospital Management System, we identified additional input fields which were absent in the Statechart model. For example, in the Patient Registration page (*Patient_Reg* in the state chart model), two additional inputs were *Name* and *Age*. For these extra input variables, the corresponding lines in

```
<mappings>
 <mapping>
  <phrase>state</phrase>
  <value>active-Patient_Reg</value>
  <webpage>Patient Registration</webpage>
  <code>"selenium.open(http://localhost:8089/"+
   "Jkek/PatientRegistrationPage)";</code>
 </mapping>
 <mapping>
  <phrase>Name</phrase>
  <value>n</value>
  <webpage>Patient Registration</webpage>
  <range>alphanumeric</range>
  <code>selenium.type("input{[]@name='name'{[]}",
    n);</code>
 </mapping>
 <mapping>
  <phrase>Age</phrase>
  <value>ag</value>
  <webpage>Patient Registration</webpage>
  <range>ag>0,ag<=30</range>
  <code>selenium.type("input{[]@name='age'{[]}",
ag);</code> }
 </mapping>
 <mapping>
  <phrase>state</phrase>
  <value>active-Patient_Login</value>
  <webpage>Patient Login</webpage>
  <code>selenium.open("http://localhost:8089/"+
    "Jkek/PatientLoginPage");</code>
 </mapping>
</mappings>
```

Figure 11: A part of Mappings XML file showing mappings between phrases used in model to Selenium RC JUnit code

the Selenium RC JUnit java code are identified and included in the mappings XML file. The domain of the values of these input variables is also identified from the implementation and included in the mapping XML file. For example, from the source code of Hospital Management System, we found that *Name* field had a constraint that it should only accept alpha-numeric values as inputs, while the *Age* field had the constraint $0<\text{age}\leq 30$. For the *Name* field which has constraint as alphanumeric in Mappings XML file, the ACT tool would generate concrete Selenium RC JUnit test cases for alphanumeric and nonalphanumeric values as test inputs. Users can use predicates in the mappings XML file. For example, for the Age field as {age>0,age≤30}, seperating conditions by commas. These constraints are included in the Mappings XML file as shown in figure 11. For the user input fields which has numeric constraints for example Age field, the ACT tool while generating concrete Selenium RC JUnit test cases, uses a numeric constraint solver Choco [17] to generate a value that satisfies all the constraints and uses that value as test inputs in the concrete Selenium RC JUnit test case.

Using the generated abstract test cases and the mappings XML file, a set of Selenium RC Junit concrete test cases is generated in this step. The ACT tool uses a numeric constraint solver Choco to generate test inputs for variables which have numeric constraints in the mapping XML file. Figure 12 shows a concrete Selenium RC JUnit test code generated for the counter example of Figure 7.

7.1 Salient Features

ACT's mapping step is different from a mechanical phrase rewriting system in the following ways:

1. **One spec, many implementations.** An interesting aspect of using the mapping technique in conjunction with the symbolic execution step is that, here, we can modify (or constrain) the specification further based on refined knowledge about the system. The same abstract test cases can be used to generate multiple test cases by adding different constraints at the mapping level. This makes it possible to use the same abstract specification to generate different sets of concrete test cases at later stages of development when the insight about the system gets refined in the form of added constraints. This approach also enables us to generate different sets of concrete test cases for different versions of the implementations from the same set of abstract test cases (and hence the same abstract specification). To do so, all we need is to define a separate mapping for each version of the system.
2. **One abstract test case, many concrete test cases.** ACT achieves a several fold increase in the number of concrete test cases w.r.t. the abstract test cases by using input space partitioning (ISP) and logic based coverage techniques. For example:
   (a) When <`range`> is `alphanumeric` (i.e. $alphabetic \vee numeric$), ACT generates a test case for all three (feasible) combinations of the two clauses.
   (b) When <`range`> is $low \leq a < high$ ($(low \leq a) \wedge (a < high)$), ACT generates 5 test cases ($(low > a)$, $(a > high)$, $(low = a)$, $low \leq a < high$, $a = high$).
   (c) We treat each input field (not already attributed concrete values at the abstract test generation stage) as a characteristic, and provide each choice coverage (ECC) for all fields in a web page.

   Graph based techniques used by the front end provide structural coverage on the specification. The above approach enhances the coverage by providing systematic coverage on the input domain.

8. RESULTS

We have implemented all the components of ACT as separate programs. As discussed, we use NuSMV model checker to generate test test paths to give node coverage (other coverage criteria are possible). Our symbolic execution engine is capable of handling basic integer arithmetic, strings and quantifier free first order predicate logic. The mapping stage handles translation of abstract test cases to concrete. Both these are implemented in Java. We used CVC3 and CHOCO SMT solvers for helping our symbolic execution and mapping steps respectively.

To evaluate our approach, we made the following measurements:

1. ability of ACT to generate abstract test cases corresponding to the test paths generated by the front end

```
@Test
public void test1()
{
  selenium.start();
  selenium.open("http://localhost:8089/Jkek/"+
   "WelcomePage");
  selenium.click("xpath=//a[contains(@href,
    'http://localhost:8089/Jkek/"+
    "PatientRegistrationPage')]");
  selenium.type("//input[@name='username']","Joe");
  selenium.type("//input[@name='password']","abc");
  selenium.type("//input[@name='name']","a12bc");
  selenium.type("//input[@name='age']","15");
  selenium.click("//input[@name='submit']");
  selenium.type("//input[@name='username']","Joe");
  selenium.type("//input[@name='password']","abc");
  selenium.click("//input[@name='checkbox1']");
  selenium.click("//input[@name='submit']");
}
```

Figure 12: A Selenium RC JUnit concrete test case for figure 7

2. the expansion in the number of concrete test cases w.r.t. the abstract test cases

As case studies, two web based enterprise applications, both developed within our institute, were used. The two web apps were:

1. Hospital Management System (HMS). This was a class project containing 133 lines of Java code.
2. Student Information System (SIS). This was a medium sized project with 15,77 lines of Java code developed using JSP/Struts.

Both applications use plain HTML without AJAX features.

Our experimental process involved the following steps:

1. We modelled the web applications using the Statechart notation presented.
2. We derived abstract test cases from Statechart model.
3. Then Selenium Remote Control JUnit concrete tests were obtained from these abstract test cases.
4. These Selenium RC JUnit concrete test cases were then executed on the implementation of the Hospital Management System (HMS) and Student Information System (SIS).

We could make the following observations:

- Simple integer arithmetic, strings and quantifier free predicate logic was found expressive enough to model all our guards and actions.
- ACT could translate all paths generated by the front end into abstract test cases. The test paths which could not be turned into abstract test cases were subsequently verified to be infeasible.

- There was a significant increase ($\times$ 4) in the number of concrete test cases w.r.t the abstract test cases.
- All concrete test cases generated successfully executed on Selenium RC.

Table 1 shows the number of nodes in the flattened Statechart (N_{flat}), the number of lines of code (LOC) of the web applications, the number of mappings in the mappings XML file, the number of abstract test cases generated for various requirements for various testing criterion, and the number of concrete Selenium RC JUnit test cases which were generated. HMS is relatively small with 133 LOC while SIS is considerably larger with 19 nodes and 15,77 LOC. A total of 14 abstract and 54 concrete test cases were generated for different test criteria for HMS. Similarly, a total of 56 abstract and 433 test cases were generated for SIS. This behavior is expected since most of the generated test cases deal with node and path coverage, which depends on the number of nodes in the flattened statechart. Furthermore, the number of lines of code has no direct relation on the number of test cases generated.

9. CONCLUSION AND FUTURE WORK

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 dependencies between atomic interactions between the (test-)client and the server. In this paper, we have provided ACT, a test generation methodology which uses symbolic execution to resolve the data dependencies between atomic interactions. Traditional symbolic execution, done along all control paths, is known to have scalability issues due to state space explosion. We perform symbolic execution along selected paths of the specification, thus avoiding state space explosion. Therefore, this approach scales to even very large specifications comprising of 100s or 1000s of states (web-pages). 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. Our mapping method, in contrast to a plain phrase to phrase translation, intelligently superimposes other black box test coverage criteria (namely, input space partitioning and logic based coverage) on the graph coverage achieved by the previous stages over the Statechart specification. This results in a meaningful expansion of the test suite during the translation of abstract test cases to concrete test cases. The test generation process is split into three main steps: *test path generation*, *abstract test case generation* and *mapping*. Each step allows introduction of additional details into the specification, facilitating stepwise refinement. This is a very realistic approach as it does not require the requirement analyst to provide a completely worked out specification in a single step. As different mappings can be used for generating concrete test cases from abstract ones, our method facilitates using the same set of abstract test cases to generate concrete test cases for different versions of the same application at a later point in time.

In our future work, we will generalise this approach to apply other forms of formal specification, coverage criteria, and front-end techniques. We will also extend this techniques to deal with more sophisticated features of web apps (e.g. AJAX).

10. REFERENCES

[1] G. A. Di Lucca, A. R. Fasolino, "Testing Web-Based Applications: the State of the Art and Future Trends", Information and Software Technology Journal, Volume 48, No 12, pp. 1172-1186.

[2] H.-J. Jo, J. G. Hwang and Y.-K. Yoon,"Formal Requirements Specification in Safety-critical Railway Signaling System", 2009 Asia Pacific Transmission & Distribution Conference, pp. 1-4.

[3] E.M. Clarke, O. Grumberg, and D. Poled. Model Checking. MIT Press, 1999.

[4] R. M. Hierons, K. Kapoor, K. Bogdanov, P. Krause,"Using Formal Specifications to Support Testing", ACM Computing Surveys, Vol. 41, No.2, Article 9, Febuary 2009.

[5] 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.

[6] F. Bolis, A. Gargantini, M. Guarnieri, E. Magri, and L. Musto, "Model-driven Testing for Web Applications using Abstract State Machines," in Current Trends in Web Engineering, Springer. Lecture Notes in Computer Science, 2012, vol. 7703, pp. 71–78.

[7] Sahi Automation Testing Tool for Web Applications: http://sahipro.com/

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

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

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

[11] J. Conallen, "Modeling Web Application Architectures with UML", Communications of ACM, Vol. 42, Issue 10, pp. 63-70.

[12] 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.

Web Applications	N_{flat}	LOC	No. of Mappings	No. of abstract tests (counter examples)		No. of concrete test cases
Hospital Management System (HMS)	6	133	20	Node coverage	6	26
				Top page is reachable from all the pages	4	13
				Every page is reachable from the top page	4	15
				Total	**14**	**54**
Student Information System (SIS)	19	15,77		Node coverage	19	161
				Top page is reachable from all the pages	17	132
				Every page is reachable from the top page	17	140
				Total	**53**	**433**

Table 1: Results of test generation on the two case studies

[13] J. C. King. "Symbolic Execution and Program Testing", Communications of the ACM, 19 (7), pp.385–394, 1976.

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

[15] A.A. Andrews, J. Offutt, R. T. Alexander, "Testing Web Applications by Modeling with FSMs", Software and System Modeling, Vol. 4, no. 3 pp. 326-345, 2005.

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

[17] T. C. D. Team, "Choco Constraint Solver", Online, 2004, http://www.emn.fr/z-info/choco-solver/.

[18] V. Santaigo, A.S.M. do Amaral, N. L.Vijaykumar, M. Mattiello-Francisco, E.Martins and O.C.Lopes,"A Practical Approach for Automated Test Case Generation using Statecharts", in COMPSAC '06, pp.183-188.

[19] R.V. Binder, "Testing object-oriented systems:models,patterns and tools", Addison-Wesley Longman Publishing Co., 1999.

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

[21] 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

[22] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An open source tool for symbolic model checking", in Proc. of 14th Conf. on Computer Aided Verification (CAV 2002), pp. 359-364.

[23] X. Jia, H. Liu,"Rigorous and Automatic Testing of Web Application", 6th IASTED International Conference on Software Engineering and Applications (SEA 2002), pp. 280-285.

[24] P. Godefroid, K. Nils, and K. Sen. "DART: Directed Automated Random Testing.", in Proceedings of the 2005 ACM Sigplan Conf. on Programming Language Design and Implementation, pp. 213-223, Volume 40, Issue 6, June 2005.

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

[26] P.Bhaduri, S.Ramesh,"Model Checking of Statechart Models: Survey and Research Direstions", ArXiv Computer Science e-prints, cs/0407038, July 2004.

[27] Selenium automation testing tool: https://www.seleniumhq.org