2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

A Testing Tool for Web Applications Using a Domain-Specific Modelling Language and the NuSMV Model Checker

Arne-Michael Törsel
Fachhochschule Stralsund, Zur Schwedenschanze 15, 18435 Stralsund, Germany
Email: Arne-Michael.Toersel@fh-stralsund.de

***Abstract*—Test case generation from formal models using model checking software is an established method. This paper presents a model-based testing approach for web applications based on a domain-specific language model. It is shown how the domain-specific language is transformed into the input language of the NuSMV model checker and how the resulting traces are converted into executable test scripts for various test automation tools. The presented approach has been implemented with comprehensive automation in a research tool which architecture is outlined.**

***Index Terms*—web applications, model-based testing, model checking, test automation**

I. Introduction

Functional web application testing on system level is usually conducted using test automation tools with tool-specific command scripts. However, manually building and especially maintaining the test suites is expensive. It can be a major problem particularly in agile development processes with 'fluid' requirements that are causing frequent adaptions of the application, as effective regression testing needs to be available all the time.

Model-based testing (MBT) is an approach to improve the efficiency of test case design. Formal models, that explicitly capture the software's desired properties, are used to derive test cases. Although test cases could manually be constructed from the models, algorithms are typically employed for automatic test case generation. This enables systematic and efficient test case generation suitable for different testing requirements. When the application requirements change during software development and the need for test case adaption arises, the models are adapted accordingly and test cases can then be generated again from the updated models. This typically requires significantly less effort than adapting the test cases.

A concretisation step is required, as, in most cases, the test cases generated from the models lack specific implementation details needed to constitute test automation tool scripts. For test process efficiency, it is important that this concretization step can be automated as well. The discussed cost of scripted test automation associated with the test suite maintenance can thus be reduced and the (regression) test effectiveness be increased as the test cases are available earlier.

Model-based testing has been applied to the domain of web application black box testing for several years [1]. Still, there are issues with the general applicability of the proposed methods and the ability for process automation which hinder the application of these methods in practical areas, as discussed in the related work section. We are presenting in this paper a model-based testing approach using a web application domain-specific language (DSL) based on extended finite state machine (EFSM) mechanisms. One advantage of such a domain-specific modelling notation is that it can be tailored exactly to the requirements and is therefore easier to learn and comprehend than generic modelling notations.

There are several principal methods for test case generation from transition-based models as EFSMs [2]. One well-known method is using the ability of model checking software to find counterexamples to claimed properties of the model [3]. The idea is to formulate a model coverage goal as a model property to be refuted by the model checker. The model checker then, if it exists, presents a counterexample trace that can be used as a test sequence.

In the approach presented with this paper, the model checker NuSMV [4] is used to produce test sequences. The automatic transformation procedure of the web application DSL to the NuSMV input language is described. Furthermore, the transformation of the traces returned from NuSMV into test scripts for various test automation tools is illustrated. The primary contribution of this paper is the description of the model based testing approach, in particular the detailed description of the transformation of the DSL into the NuSMV input language.

The rest of the paper is organized as follows. Section II contains an overview about the model-based testing workflow. Section III reviews the domain-specific language used for modelling web applications that was introduced in an earlier paper [5]. The transformation of the domain-specific language to the NuSMV input language is described in detail in Section IV. Section V illustrates the presented concepts with excerpts of a case study applying the model based testing approach to a real world application. Finally, in section VI related work is reviewed and its relation to the work presented here is discussed.

II. Model-based testing workflow

This section outlines the model-based testing workflow that has been implemented as a research tool[1] using the Java programming language and Eclipse Modelling Framework/Xtext

[1] http://www.mbt4web.fh-stralsund.de

978-0-7695-4968-2/13 $26.00 © 2013 IEEE
DOI 10.1109/ICST.2013.54

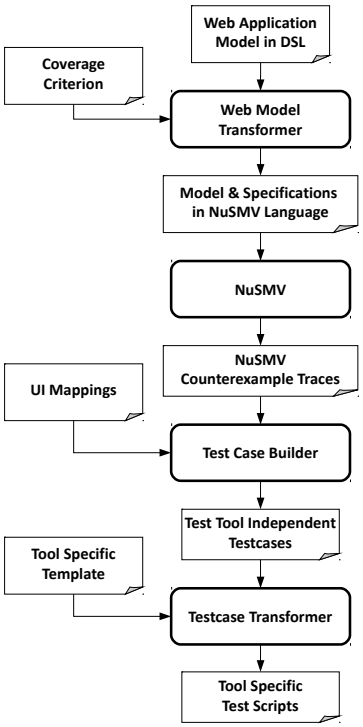

Fig. 1. Schema of model based testing workflow

technology[2]. We have designed a model-based black box testing approach for web applications with extensive automation along the whole process from the model to executable test cases. This process is depicted in Figure 1.

The process starts with building the web application model using the web application DSL. The primary reason for choosing a textual modelling notation over a graphical one for the research tool is that comfortable editor support could be achieved without much effort using Eclipse Xtext. There is also some indication in the literature [6], that textual modelling is more efficient for model-based testing purposes than graphical modelling. Nevertheless, a complementary graphical editor could be implemented quite easily using the Eclipse Graphical Modelling Framework (GMF) as well.

To generate test cases, a coverage goal has to be selected by the user. There are popular criteria as for example covering all transitions or trying to build a tour over all states. The DSL and the coverage criterion are translated into a NuSMV model. The Kripke structure is constructed from the DSL and the coverage criteria selected by the user are translated into temporal logic expressions (either Computation Tree Logic/CTL or Linear Temporal Logic/LTL, depending on the criterion) to be checked. This is explained in detail in section IV of the paper.

NusMV is executed and the generated specifications are checked. The resulting counterexample traces are converted into test tool independent test cases. At this point application implementation specific information that were missing from the DSL for abstraction reasons are integrated into the test cases. This is done using an UI mapping file that serves to map abstract identifiers from the model to concrete information needed for test execution as for example HTML Id‘s or XPath expressions to locate web page elements or URLs of web pages that need to be accessed directly.

Using a template for the targeted test automation tool, test scripts are generated from the tool independent test cases. Therefore this intermediate test case format serves to decouple the test case generation from a specific tool. New test tools can be supported simply by adding templates. At the moment, the research tool supports the two popular tools *Selenium 2* and *Canoo WebTest*. After execution of the test cases, if there has been an error, the cause needs to be traced back to model. This is currently supported by meta information that are inserted into the generated test scripts.

III. The Domain-Specific Language

Considering the target to generate executable test cases for test automation tools with a high degree of automation, four principal requirements for a modelling language were identified:

- Modelling the structure of a web application: this is achieved with the concept of *Views* and user caused *Transitions* to other views. A transition from one view to another view is triggered by the user, for example by clicking a link, a button or even an arbitrary element.
- Expressing dependencies between workflows in the model: in analogy to the EFSM mechanisms, this is achieved by introducing model variables that serve as guards for transitions and can be assigned values as an effect of transitions.
- Referencing external test data: this is realized in the form of key-value containers through a special data type *Tuple*. Notice that input data is not generated in this approach but always supplied from either primitive model variables or external data sources using the *Tuple* data type.
- Abstracting from implementation specification data as for example HTML Ids: as explained in the previous section, this is important to keep the model abstract and easy to understand. Furthermore, when data as an HTML button Id changes, the necessary adaptation has to be made only once in the UI mapping file not multiple times in the model file. This benefits the maintainability.

Listing 1 shows an example with all language constructs that play a role in the transformation into the NuSMV input language, Figure 2 shows a graphical representation of the modelled scenario. The example defines four views using the keyword *View*: Start, Login, Profile and News. The keyword *Transition* is used to declare user triggered transitions to other views within a *View* block.

Because of the guard condition stated by the keyword *When* at line 9, the Profile view is only accessible after the user has logged in in the Login view. At line 2, the referenced

[2]http://www.eclipse.org/Xtext

```
1  Tuple user = "UserDataSource"
2  Bool loggedIn = false

4  View Start {
5    Text "Welcome [user.username]" When loggedIn == true
6    Text "Please log in" When loggedIn == false

8    Transition Login {
9      When loggedIn == false
10     Link "login.link"
11   }

13   Transition Profile {
14     When loggedIn == true
15     Link "profile.link"
16   }

18   Transition News {
19     Link "news.link"
20   }

22 }

24 View Login {
25   Text "Please enter login information"

27   Transition Start {
28     Input "username.input" = "[user.username]"
29     Input "password.input" = "[user.password]"
30     Button "login.submit"
31     Assign loggedIn = true
32   }
33 }

35 View Profile {
36   ...
37 }

39 View News {
40   ...
41 }
```

Listing 1. Model example source code

Boolean variable 'loggedIn' is declared. In this example it stores, whether or not the user is logged into the application. The variable is set from its initial value 'false' to 'true' as part of the transition from the Login view back to the Start view (lines 27-32) using the keyword *Assign* at line 31. Once the user has logged in, the Login view is no longer accessible because of the guard condition at line 14. The News view is always accessible from the Start view because no guard condition is attached to the transition from the Start view to the News view at lines 18-20.

All three transitions from the Start view are triggered by the user clicking a link - this is expressed by the keyword *Link*. The parameter to this keyword is a reference key that is looked up from the UI mapping file to get the implementation specific information later during test case creation. The transition from the Login view to the Start view (lines 27-32) is modelled as a web formular, where input from the user is received. Here, two input fields are filled with test data from the *Tuple* variable 'user', but static data could be used here as well. In this case, a button has to be clicked to trigger the transition. In the same manner as with the links, input elements and buttons are referenced by abstract keys to be looked up later from the UI mapping file. Listing 2 shows an excerpt of an UI mapping file for the example model. The UI mapping file is a simple

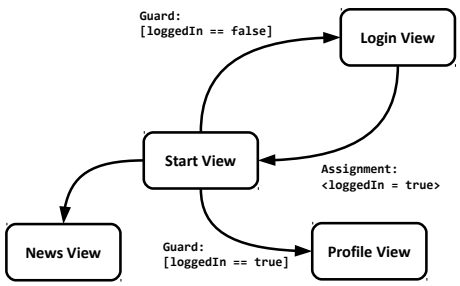

Fig. 2. Graphical representation of the example

```
<mbt4web>
  <view name="Start"
    url="http://localhost/forum"/>
  ...
  <link name="profile.link" reftype="url"
    refdata="/profile"/>
  ...
  <input name="username.input" reftype="id"
    refdata="username"/>
  ...
  <button name="login.submit" reftype="xpath"
    refdata="//input[@type='submit']"/>
</mbt4web>
```

Listing 2. UI mapping file for the model example

XML-based file format that is referenced by the DSL model through the *MappingFile* keyword.

The *Text* directives in lines 5 and 6 serve as test oracles. When building test scripts, they are converted into test tool-specific commands that try to locate the given text in the web page. The text expression may be dynamic by including variable values.

There are several other features of the DSL not covered in the listing source code, for example:

- Arbitrary sophisticated test oracles can be produced using pre-fabricated XPath expressions stored in the UI mapping file that can be parameterized during test case generation with model variable values. In the generated test script, a command is inserted that will evaluate the web page, checking for the existence of elements matching the XPath expression.
- The model source code can be distributed into multiple files using an import mechanism to improve the manageability of the model. In the editor, references are then resolved across files.
- Through the *Forward* keyword, conditional redirects of the web application can be modelled. This is explained in more details in the next section.

IV. Construction of the NuSMV model

A. Converting the DSL

This section describes how the application model DSL is transformed into the NuSMV input language. Section V

contains a comprehensive example case study for further clarification. The described transformation process is performed automatically without any user assistance being required in the research tool.

Scalar variables named 'view' and 'trans' are created for all *View* and *Transition* elements of the application model. The 'view' variable contains the names of all the views as symbolic values, the 'trans' variable contains generated unique names as symbolic values for all transitions in the model. Both variables are augmented with a symbolic value named *dead_end_marker*. Using the NuSMV *init* constraint, the 'view' variable is initialized to the view named 'Start' and the 'trans' variable is initialized to the value *dead_end_marker*.

All boolean variables of the application model DSL are mapped to boolean variables in the NuSMV input language, with their initial value set to the value given in the application model. The generator keeps track of assignments to variables of the other types supported by the DSL, *String* and *Number*, using a separate mechanism. The reason only the *Boolean* model variables are mapped to NuSMV is that only they are used in model *When* expressions that are converted to NuSMV *next* expressions, as shown in the next paragraphs. The inability to use other variable types, in particular numeric variables, in guard expressions is a current limitation of the described approach. As this would lead to a considerably higher complexity with a definite negative impact on the scalability using only NuSMV, an extension of the method using technologies as SMT solving may be a topic of further research. For example, Bentakouk et al. [7] describe the application of the Z3 solver for web service testing.

Usually, when converting state machines to NuSMV models, one variable is associated with the current state. Inside the *next* expression for this variable, a *case* expression is constructed where there is one *case* clause for every state that lists all possible subsequent states. This is not possible when converting the discussed web application model. Imagine a hypothetical situation with a variable named 'state', where there are two possible subsequent states from the value 's1'. One transition to the state 's2' is unconditional, the other one to state 's3' is guarded. There would have to be two *case* clauses to be modelled in the *case* expression:

```
next(state) :=
case
  state = s1 : s2;
  state = s1 & condition = true : s3;
```

Let us assume, that the guard condition is fulfilled. In this case, NuSMV would still always pick the first matching *case* clause, selecting 's2' as the next value for variable 'state'.

To circumvent this problem, another solution is used in the research tool. In the NuSMV model a *next* expression for the transition variable 'trans' is inserted. One *case* clause for every view is constructed listing all transitions leaving that view as in this example:

```
next(trans) :=
case
  view = Start : {list of outgoing transitions}
  view = ... : {list of outgoing transitions}
```

At this point, the guard conditions are not considered.

The *next* expression for the 'view' variable is generated by creating one *case* clause for every transition. If a transition contains a guard condition expression (declared by the keyword *When* in the model DSL), this expression is translated to a NuSMV boolean expression and is added to the clause condition, as illustrated in the following example:

```
next(view) :=
case
  next(trans) = transition_1 : View2
  next(trans) = transition_2 & <conditions> : View3
  ...
  view = dead_end_marker: Start;
  TRUE : dead_end_marker;
esac;
```

In the default case, having the *case* condition *TRUE*, the 'view' variable is set to the value *dead_end_marker*, leading to an infinite loop in this branch of the computation tree that is detected by NuSMV. This is intentional. If a transition was selected in the *next* expression for the 'trans' variable which condition is not fulfilled, this choice is invalid.

To deal with views that do not have any outgoing transition, a further flag value *reset_marker* is added to the 'trans' variable scalar. It enables the model checker to revert to the 'Start' view if a view is reached, where no outgoing transition is available. This is implemented as the default case in the *next* expression for the variable 'trans':

```
next(trans) :=
case
  ...
  TRUE : reset_marker
esac
```

Besides, in the *next* expression for the 'view' variable a *case* clause selecting the view 'Start' is added as first case:

```
next(view) :=
case
  next(trans) = reset_marker : Start;
```

Another concept important to web application testing is forwarding through conditional redirects. Imagine an application the user can log into. After logging in, whenever the user enters the start page's URL, the application transparently redirects to a personal dashboard. There is no direct transition from the start page to the dashboard. This situation is accounted for by introducing further rules that are generated with the *Forward* keyword in the web application model. The model DSL syntax inside *View* blocks is:

```
Forward <forward_view_name> When <forward_condition>
```

There may be multiple forward rules inside *View* blocks. During the conversion into the NuSMV language, a *Forward* is transformed as follows: in the *next* expression for the 'view' variable a *case* clause is added, that will select the forward target *View*, given that the current *View* is the forward source and the forwarding condition is met.

```
next(view) :=
case
  ...
  view = <source_view_name>
    & <forward_condition> : forward_view_name;
```

In the *next* expression for the 'trans' variable, also a *case* clause is added, which will set the next value of the variable to *dead_end_marker*, again given that the current *View* is the forward source and the forwarding condition is met.

```
next(trans) :=
case
  ...
  view = <source_view_name>
    & <forward_condition> : dead_end_marker;
```

Assigning *dead_end_marker* as value for the 'trans' variable in case of a forwarding allows the test case transformer to recognize the forwarding. The test case transformer will then generate test script steps to verify that the forwarding target view is displayed instead of the forwarding source view.

B. Converting the Coverage Criteria

This section describes how the coverage criteria selected by the tool user are translated into CTL/LTL specification formulas in the NuSMV input language. The tool currently offers to:

- Cover views or transitions separately. This means for every target a single test case containing all necessary precondition steps is constructed. That is why there often is a lot of test step repetition in the test cases. Imagine an application where most work flows depend on the user being logged in - the necessary login steps will be included in every generated test case that depends on the login.
- Cover groups of views or transitions (possibly all) in a single test case. Such test cases can only be constructed if none of the two single goals are mutually exclusive.

The first coverage criterion is converted to the NuSMV input language using a CTL expression. An example is shown in the following excerpt where all views from the model in Listing 1 shall be covered separately (except the Start view which is implicitly covered because it is the application entry point).

```
CTLSPEC NAME reach_Profile := ! (EF view = Profile);
CTLSPEC NAME reach_News := ! (EF view = News);
CTLSPEC NAME reach_Login := ! (EF view = Login);
```

The CTL operator *EF* is used to express that there exists at least one evolution of the system where the 'view' variable eventually holds the value of the particular *View* name, e. g. 'Profile' in the first line. As the expressions are negated using the exclamation mark operator, the CTL formulas express the fact that there is no evolution of the system where that particular view is reached. By constructing a counterexample trace, NuSMV provides the generator with a sequence covering that view. This principle can be applied for covering single transitions in the very same way.

The second criterion is difficult to express using a CTL formula. Because CTL is a branching time logic, it considers all possible system evolutions from a given point in time. LTL, in contrast, is a linear time logic. It considers specific evolutions of a system from a given point in time [8]. Because for this coverage criterion NuSMV has to generate a counterexample trace by refuting the claim that all goals can not be met within a single evolution of the system, it is constructed using an LTL expression. The following example shows the criterion to cover all views (again, except the Start view) from Listing 1 in a single sequence as an LTL expression. Note that the order in which the views are covered by the sequence is arbitrary and not implied by the expression.

```
LTLSPEC NAME reach_all_views :=
  ! F(O view = Profile & O view = News
    & O view = Login);
```

The LTL operators *F* and *O* are used in this expression. With the operator O is stated that a fact - in this case that the variable view holds a specific *View* name value - is true at least once in a specific evolution of the system. Then the operator *F* is applied to the expression inside the braces to state that this shall hold at some point in the future. Again, by negating the whole expression with the exclamation mark operator, it is stated that there is no evolution of the system where the 'view' variable has held all the given view name values (Profile, News, Login) at least once. The counterexample presented by NuSMV then yields a sequence with all given elements covered in a valid order (if possible). As with the first criterion, this method can be applied to formulate transition coverage goals as well. More complex coverage criteria could be realized in the research tool. This is an area of ongoing enhancement. For example, using the ability to express sequences of variable values in the LTL, the tester could specify sub paths that shall be contained inside a generated test case to cover 'typical' work flows.

V. Case Study

This section illustrates the modelling and transformation approach with a model excerpt for testing the popular open source bug tracking application Mantis[3]. For the case study, core workflows as for example logging in, opening a bug, closing a bug and commenting a bug have been modelled using the discussed domain-specific language. Figure 3 displays the navigational structure of the model consisting of views and transitions. In the used Mantis configuration, specific actions have to be executed by different user roles, therefore the guard conditions in the model mainly express transition dependencies on the logged-in user account. Listing 3 shows excerpts of the domain-specific model with clarifying comments.

Listing 4 shows excerpts of the model in the NuSMV input language that was automatically created from the domain-specific language model in Listing 3. Note that the variable values for the scalar transition variable 'trans' are built using the format "(start view name)_(target view name)_(unique identifier number)" to cope with multiple parallel transitions between two views. Listing 5 shows a generated test case in an intermediate format, which is the input for the transformation into test tool specific scripts. The test target that led to creation of this test case was to reach the view 'ReportIssueConfirmation'. The corresponding CTL specification is displayed in the last line of Listing 4. Notice that, while the commands in

[3]http://www.mantisbt.org/

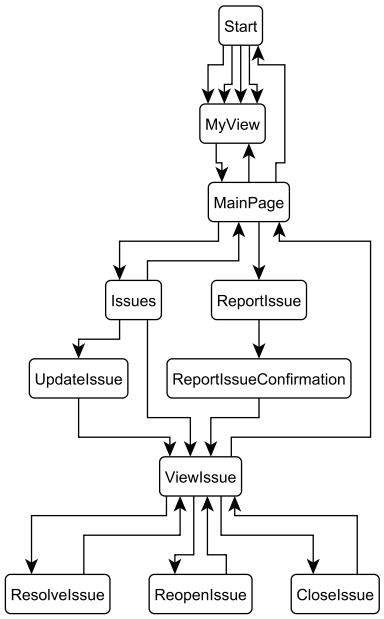

Fig. 3. Structural view of case study model

the test case description format in Listing 5 are abstract, the UI specific information from the UI mapping file is already contained, i.e. the format is test tool independent but not abstract in the same way as the model is. The transformation modules in the research tool are programmed in the Xtend language[4] but other languages running on the Java VM could be used for developing transformation modules to support further test automation tools as well.

For the study, the model was built incrementally, creating small parts and verifying the resulting test cases against the already existing application. This way modelling errors could be identified quickly. One lesson learned from this study therefore is that in development projects, it is advisable to build the model at the same time as the application. This way cross-checks can be performed to ensure both the validity of the application and the model, with respect to the requirements. A test-driven development approach is also feasible because the model can be created prior to the software since the implementation details are isolated in the UI mapping file. Clearly, the initial modelling effort further pays off if the generated test suite is used for regression testing, especially if it has to be maintained due to requirement changes. To gather more experience with the usability of the tool in real life development projects, more case studies are planned where the development process is accompanied by the tool.

VI. Related Work

Similarly to this paper, Li et al. [9] describe a testing approach for web applications that utilizes the NuSMV model checker for test case generation. Homma et al. [10] use the

[4]http://www.eclipse.org/xtend/

```
\\ Reference the UI mappings
MappingFile "mantis-config.xml"
...
\\ reference external test data in property files
Tuple userdata = "mantis-userdata.properties"
Tuple bugdata = "mantis-bugdata.properties"
...
\\Variable declaration, parts omitted
Chars lastTicketSummary = ""
Chars lastTicketDescription = ""

Bool managerLoggedIn = false
...
Bool bug_reported = false
Bool bug_open = false
...

\\ Views declaration, parts omitted
View Start {
  Forward MyView When (adminLoggedIn == true |
    managerLoggedIn == true | developerLoggedIn == true
    | reporterLoggedIn == true)

  Title "MantisBT"
  Text "allow your session to be used from this IP"

  \\ Example for manager role
  Transition MyView {
    Input "username" = "[userdata.managerUser]"
    Input "password" = "[userdata.managerPass]"
    \\ 'login' is resolved from UI mapping file
    Click "login"
    Assign loggedInUser = "[userdata.managerUser]"
    Assign managerLoggedIn = true
  }

  ....
}

...

View Issues {
  Title "View Issues - MantisBT"
  Text "[lastTicketSummary]" When bug_reported == true

  Transition MainPage {
    Click "main"
  }

  Transition ViewIssue {
    When bug_reported == true
    Click "viewfromtable"
  }

  Transition UpdateIssue {
    When managerLoggedIn == true & bug_reported == true
    Click "updatebug"
  }
}

...

View ReportIssue {

  Transition ReportIssueConfirmation {
    Input "bugcategory" = "[bugdata.category]"
    Input "bugsummary" = "[bugdata.summary]"
    Input "bugdescription" = "[bugdata.description]"
    Click "submitbug"
    Assign lastTicketSummary = "[bugdata.summary]"
    Assign lastTicketDescription
      = "[bugdata.description]"
    Assign bug_reported = true
    Assign bug_open = true
  }
}
```

Listing 3. Case study model excerpt

```
VAR
  trans : {Start_MyView_21279119, ...,
    ReportIssue_ReportIssueConfirmation_22744620,
    ..., dead_end_marker, reset_marker};
  view : {Start, MyView, MainPage, Issues,
    ..., dead_end_marker};
  managerLoggedIn : boolean;
  ...
  bug_open : boolean;
  bug_reported : boolean;
  ...

ASSIGN
init (view) := Start;
init (trans) := dead_end_marker;
init (managerLoggedIn) := FALSE;
...
init (bug_open) := FALSE;
init (bug_reported) := FALSE;
...

next (trans) :=
case
  view = dead_end_marker : dead_end_marker;
  view = Start &
    (managerLoggedIn = TRUE |
    developerLoggedIn = TRUE |
    reporterLoggedIn = TRUE)
    : dead_end_marker;
  view = Start : {
    Start_MyView_21279119, Start_MyView_16291022,
    Start_MyView_17226426, Start_MyView_16278782};
  ...
  view = ReportIssue :
    ReportIssue_ReportIssueConfirmation_22744620;
  ...
  TRUE : reset_marker;
esac ;

next (managerLoggedIn) :=
case
  next(trans) =
    Start_MyView_16291022 & next(view) = MyView
    : TRUE;
  next(trans) =
    MainPage_Start_13044493 & next(view) = Start
    : FALSE;
  TRUE : managerLoggedIn;
esac ;

...

next (view) :=
case
  next(trans) = reset_marker : Start;
  view = Start & (managerLoggedIn = TRUE |
    developerLoggedIn = TRUE |
    reporterLoggedIn = TRUE) : MyView;
  next(trans) = Start_MyView_21279119 : MyView;
  ...
  next(trans) =
    ReportIssue_ReportIssueConfirmation_22744620 :
    ReportIssueConfirmation;
  ...
  TRUE : dead_end_marker;
esac ;
...

CTLSPEC NAME Reach_ReportIssueConfirmation :=
  ! (EF view = ReportIssueConfirmation);
}
```

Listing 4. Excerpts of the model in NuSMV language

```
InvokeURL "http://localhost:8080/mantis"
VerifyTitle "MantisBT"
VerifyPageContent
  "allow your session to be used from this IP"
SetInputField name "username" = "webmbt-reporter"
SetInputField name "password" = "admin4u"
ClickButton xpath "//input[@type=\'submit\']"
VerifyTitle "My View - MantisBT"
VerifyPageContent "Logged in as:"
VerifyPageContent "webmbt-reporter"
ClickLink url "main_page.php"
VerifyTitle "Main - MantisBT"
ClickLink url "bug_report_page.php"
SetSelectField name "category_id" = "1"
SetInputField name "summary" =
  "Test Ticket Summary"
SetInputField name "description" =
  "Test Ticket Description"
ClickButton label "Submit Report"
VerifyPageContent "Operation successful"
```

Listing 5. Generated test tool independent test case

SPIN model checker to both verify web application in terms of page reachability and to generate test cases. Their approach is based on modelling the application by two automata: one for the page transitions and one for the internal application state transitions which are synchronized by message passing. The state transitions can be predicated by variables. Both approaches seem to target the manual construction of the model in the respective input language of the used model checker. This may be cumbersome and error-prone for all but the smallest applications. Hallé et al. [11] focus on errors caused by navigational browser input and use a state machine to constrain the allowed navigation sequences. NuSMV is used to verify this state machine.

Andrews et al. [12] used finite state machines for web application testing, but their models do not incorporate the necessary logic and data flow information for automatic construction of test scripts. Also their model does not cope with conditional redirects, as the proposed method does. Marchetto et al. [13] present an approach that utilizes recorded user interaction data to construct a state machine model especially for testing AJAX functionality. 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.

Another approach that targets testing AJAX-based web applications is presented by Mesbah and van Deursen [14]. Instead of real users providing execution traces, a special crawler is used here to build a model. To provide input when necessary, the crawler is backed by a database of input data that is associated with hash values calculated from the web page form attributes. It remains unclear how the crawler selects the correct set of input data dependent on both the web page form and the current application state. A test oracle is provided by invariant expressions on the DOM tree. Generic invariants like syntactic validity can be complemented by application specific invariants stated as XPath expressions. A related approach that also relies on crawling to build an application model is implemented in the research tool *WebMate* [15].

Necessary input data that cannot be randomly generated, as for example login data, has to be provided to the crawler. The authors state this to be the major weakness of that approach.

Ernits et al. [16] describe the usage of the *NModel* tool for web application testing. The C# language is used to construct a finite state machine as a model program. An adaptation layer contains test oracle code for both front end and back end testing, which remedies the problem of low observability of test impact pertinent to web application black box testing [12]. As interaction and verification functionality have to be implemented in the adaptation layer, the authors stress that more effort is needed to implement this layer than for the application model itself. Wang et al. [17] use a directed graph to model the navigation paths of the web application. Test sequences that cover all pairwise interactions between web pages are generated. However, the dependency of feasible navigation paths on the application state is not modelled.

With an initial focus on test case prioritization, Bryce et al. describe their first efforts [18] towards a unified model and terminology for both general GUI applications and web applications building upon the event flow model proposed by Memon [19]. To derive test cases from the event flow model, several 'event space exploration strategies' can be employed. Information about the state transformation of GUI components is used for test oracle generation. In comparison to the proposed method, the dependency of application behaviour on test input data is not explicitly expressed, because the model is to be automatically constructed using a user interface crawler.

VII. Conclusion

We have presented a highly automated model-based testing approach for web applications that has been implemented in a research tool. It is suited to contribute test scripts to a regression test suite where baseline scenarios are generated from models and more intricate test scenarios are manually designed by the test engineer. This should be useful especially in agile development processes, because these baseline scenarios could constitue a smoke test suite that quickly reveals obvious problems. Also, a test driven software development approach is possible, where the model is built prior to the software.

Planned enhancements will improve the applicability in real world projects. One important enhancement for the domain-specific language is the ability for hierarchical model composition as implemented by other methods [12]. This is an interesting research problem, because a simple flattening of the models could lead to very large models that may not be processible by NuSMV within reasonable amounts of time and memory. Another planned extension is the inclusion of user-specified sub-paths in generated test sequences as described in section IV.

Acknowledgment

This work was funded by Deutsche Forschungsgemeinschaft (DFG) grant BL 891/2-1.

References

[1] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Modelling methods for web application verification and testing: state of the art," *Softw. Test. Verif. Reliab.*, vol. 19, pp. 265–296, 2009.

[2] A. Pretschner, M. Utting, and B. Legeard, "A taxonomy of model-based testing," Department of Computer Science, University of Waikato, Tech. Rep., 2006.

[3] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-7. London, UK, UK: Springer-Verlag, 1999, pp. 146–162.

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking," in *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Copenhagen, Denmark: Springer, July 2002.

[5] A.-M. Törsel, "Automated test case generation for web applications from a domain specific model," *Computer Software and Applications Conference Workshops*, vol. 0, pp. 137–142, 2011.

[6] A. Sinha, C. E. Williams, and P. Santhanam, "A measurement framework for evaluating model-based test generation tools," *IBM Syst. J.*, vol. 45, no. 3, pp. 501–514, Jul. 2006.

[7] L. Bentakouk, P. Poizat, and F. Zaïdi, "Checking the behavioral conformance of web services with symbolic testing and an smt solver," in *Proceedings of the 5th international conference on Tests and proofs*, ser. TAP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 33–50.

[8] M. Y. Vardi, "Branching vs. linear time: Final showdown," in *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS 2001. London, UK, UK: Springer-Verlag, 2001, pp. 1–22.

[9] L. Li, H. Miao, and S. Chen, "Test generation for web applications using model-checking," in *Software Engineering Artificial Intelligence Networking and Parallel/Distributed Computing (SNPD), 2010 11th ACIS International Conference on*, june 2010, pp. 237 –242.

[10] K. Homma, S. Izumi, K. Takahashi, and A. Togashi, "Modeling, verification and testing of web applications using model checker," *IEICE Transactions*, pp. 989–999, 2011.

[11] S. Hallé, T. Ettema, C. Bunch, and T. Bultan, "Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 235–244.

[12] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing web applications by modeling with fsms," *Software and System Modeling*, vol. 4, no. 3, pp. 326–345, 2005.

[13] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 121–130.

[14] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 210–220.

[15] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "Webmate: a tool for testing web 2.0 applications," in *Proceedings of the Workshop on JavaScript Tools*, ser. JSTools '12. New York, NY, USA: ACM, 2012, pp. 11–15.

[16] J. Ernits, R. Roo, J. Jacky, and M. Veanes, *Testing of Software and Communication Systems*. Springer, 2009, ch. Model-Based Testing of Web Applications Using NModel, pp. 211–216.

[17] W. Wang, S. Sampath, Y. Lei, and R. Kacker, "An interaction-based test sequence generation approach for testing web applications," *High-Assurance Systems Engineering, IEEE International Symposium on*, 2008.

[18] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a single model and test prioritization strategies for event-driven software," *IEEE Transactions on Software Engineering*, vol. 37, pp. 48–64, 2011.

[19] A. M. Memon, "An event-flow model of gui-based applications for testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.