A Versatile Tool Environment to Perform Model-based Testing of Web Applications and Multilingual Websites

Winfried Dulz
TestUS Consulting, Nuremberg, Germany

Keywords: Model-based Testing, Markov Chain Usage Model, Test Suite Generation, Test Suite Assessment, Model-based Visualization, Website Testing, Selenium.

Abstract: This paper examines techniques for the model-based testing of web applications and multilingual websites. For this purpose, the simple web application `HelloMBTWorld` is used to explain the essential steps for a model-based test process that applies statistical usage models to generate and evaluate appropriate test suites. Model-based techniques that provide graphical representations of usage models allow to set the test focus on specific regions of the system under test that shall be tested. Based on adopted profiles different user groups can be distinguished by different test suites during the test execution. Generic usage models, which are adjusted to specific language environments during the test execution, permit the testing of multilingual websites. Using the Eclipse modeling framework in combination with the TestPlayer tool chain provides a versatile tool environment for model-based testing of web applications and websites.

1 INTRODUCTION

Developing complex software and embedded systems usually consists of a series of design, implementation and test phases. Due to the increasing complexity of networked systems, for example for IoT (Internet of things) applications, model-based development approaches are becoming increasingly popular. Each software engineering step is guided by a suitable method and is usually supported by a special tool.

1.1 Model-based Testing

One method in which the test cases are generated from a model is called *Model-based Testing* (El-Far and Whittaker, 2001), (M. Utting, 2007). The relationship between a model that describes those parts of a given SUT (*system under test*) that need to be tested in order to generate (automatically) test cases derived from the (graphical) model is illustrated in Fig. 1. In general, a distinction is made between

- *system specifications*, which model functional or non-functional aspects of the SUT and
- *usage models* that describe the usage behavior of the future users of the system when they interact with the SUT in different ways.

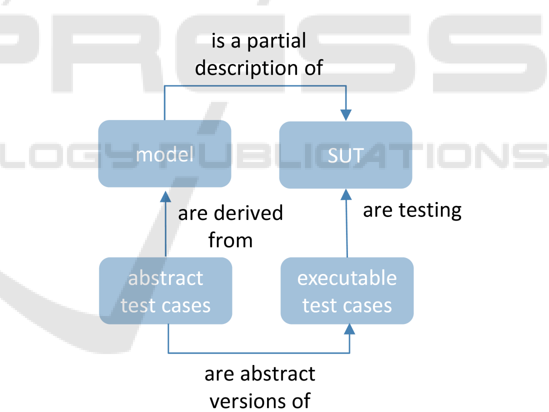

Figure 1: General approach of Model-based Testing.

Test cases, which are generated from a system specification (Rosaria and Robinson, 2000) are often used in the so-called component or unit test. Usage models are mostly applied to generate test cases for the system or acceptance test. The popular *V-Model* (Tian, 2005) illustrates this relationship by distinguishing between the development and test phases of a system during the systems engineering process.

1.2 Statistical Testing

Since complete testing of real systems is not feasible in practice, a suitable set of test cases must be selected to achieve a specific test objective. With the help

Dulz, W.
A Versatile Tool Environment to Perform Model-based Testing of Web Applications and Multilingual Websites.
DOI: 10.5220/0006838400450056
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 45-56
ISBN: 978-989-758-320-9
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

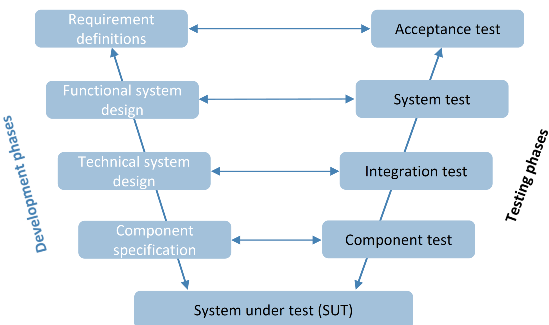

Figure 2: *V-Model* (Tian, 2005) for Software Engineering.

of statistical usage models, also called *Markov chain usage models* (MCUM) (Whittaker and Poore, 1993), (Walton et al., 1995), individual test cases or complete test suites can (automatically) be derived by traversing the MCUM. Markov chains are graphical models to define all possible usage steps and scenarios on a given level of abstraction as shown in Fig. 3. MCUM are used to represent

- *usage states* for modeling the user behavior during the interaction with the system, as well as
- *state transitions* to specify the reaction of the system on a user's interaction.

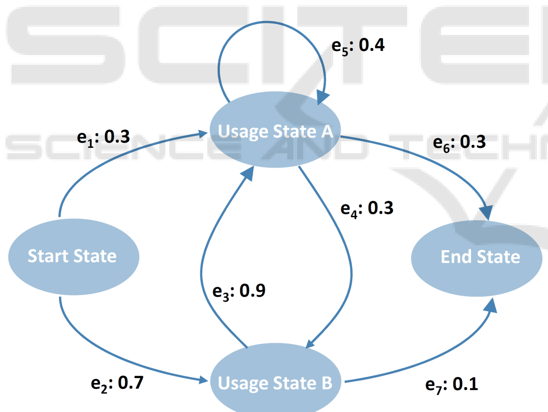

Figure 3: Statistical Markov chain usage model for modeling the usage behavior of system users.

The probability that a particular user interaction triggers an event e_j is called *transition probability* and is given behind a colon, e.g. $e_4 : 0.3$ to change from `Usage State A` into `Usage State B` (Fig. 3). By adjusting the probability values of the usage distribution, i.e. the *operational usage profile* (Musa, 1996), it is easy to specify a varying usage behavior for different *user classes*. In this way, the test engineer can automatically create distinct test cases for different system users.

A *test case* is given by a statistical traversal of the Markov chain beginning in the `Start State` and ending in the final `Stop State`, considering the probabilities of the selected usage profile. A *test suite* is a set of test cases to achieve a specific test objective, e.g. to cover all usage states or to traverse all transitions at least once during the test execution. How to derive the usage distribution for a Markov chain usage model in a more systematic way is discussed in (Whittaker and Poore, 1993), (Walton and Poore, 2000), (Poore et al., 2000), (Takagi and Furukawa, 2004) and (Dulz et al., 2010).

The main goals when using a MCUM for statistical test case generation can be summarized in

- automatic *generation* of sufficient many test cases
- calculation of meaningful *metrics* for the test suite
- determine *stopping criteria* for terminating the test execution.

Interesting test metrics can be calculated immediately after the test suite generation, e.g.

- mean number of test cases that are necessary to cover all states and transitions of the Markov chain (*steady state analysis*)
- probability for the occurrence of a certain state/transition in a given test case (*feature usage*)
- mean length of a test case to estimate the test duration (*source entropy*)

and after the test execution the number of test cases that passed and failed the test or the *reliability* of the SUT can be presented.

Computations for Markov chain usage models are the result of years of work by many different people and summarized by Stacy Prowell in (Prowell, 2000). James Whittaker and Jesse Poore did the original work on Markov chain usage models (Whittaker and Poore, 1993). Gwen Waltons research applied mathematical programming techniques to set model probabilities under testing constraints (Walton and Poore, 2000). Jenny Morales and Dave Pearson investigated combining information across tests to improve reliability measurements (Prowell, 2000). Kirk Sayres research provided many new and useful analytical results, and provided stopping criteria for statistical testing (Sayre and Poore, 2000). Walter Gutjahr demonstrated how a Markov chain could be modified to bias test generation toward low-use critical function, and how the bias could be removed in the results (Gutjahr, 1997).

Our expertise and experience from various national and international projects, such as reviewing the recent ISTQB Foundation Level Model-Based Tester Syllabus[1], also show that Markovian usage

[1] http://www.istqb.org/downloads/category/6-model-based-tester-extension-documents.html

models are very well suited for the testing of various applications in a broad range of domains such as information and communication technology, medical technology, automotive and automation technology.

1.3 Automated Statistical Testing

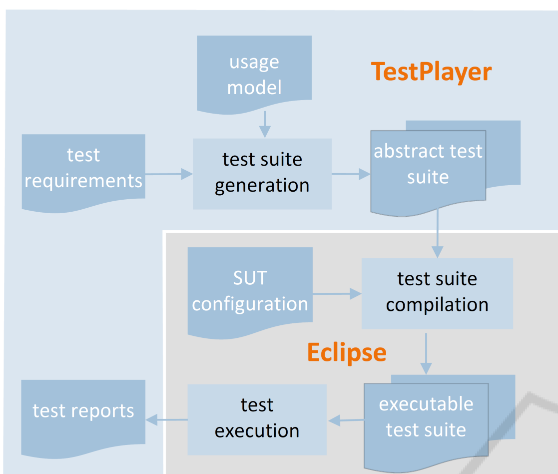

Figure 4: A versatile tool environment consisting of the TestPlayer and Eclipse.

Given the promising properties and results of Markovian usage models, there is a demand to provide a suitable tool environment for automating the test case generation and test execution process. As far as we know, there are just two other tools besides the TestPlayer©, developed by TestUS[2] that are focused on the testing with statistical usage models.

A more scientifically oriented research platform called JUMBL[3] (Prowell, 2003) was developed at the SQRL (Software Quality Research Laboratory, University of Knoxville Tennessee) and the commercial tool MaTeLo (Dulz and Zhen, 2003) is licensed by ALL4TEC[4].

Compared to the other tools, the TestPlayer provides a variety of useful graphical representations to evaluate the properties of automatically generated test suites and to decide which and how many test cases are needed to achieve a particular test objective (Dulz, 2011). As illustrated in Fig. 4, the Eclipse modeling platform is well suited for compiling an *executable* test suite and for performing the test execution after the TestPlayer has generated an *abstract* test suite from a given usage model and additional test requirements.

In the next sections we take a closer look on the underlying test case generation processes shown in Fig. 4. Using a simple web application and the corresponding user model, we demonstrate how typical tasks in the modeling, test case generation, analysis and the selection process can be performed. We will also examine some metrics that will enable us to control the selection of the test suite and to decide which test cases are best suited to meet certain test requirements.

[2] https://www.testus.eu/
[3] http://jumbl.sourceforge.net
[4] http://www.all4tec.net/

2 TESTPLAYER - A TOOL FOR AUTOMATIC TEST CASE GENERATION

The TestPlayer can be executed in any modern web browser via a graphical user interface based on RESTful web technologies. Specific elements in the TestPlayer Dashboard (Fig. 5) allow a comfortable and user-friendly control of all sub-tasks, which must be performed for statistical usage tests.

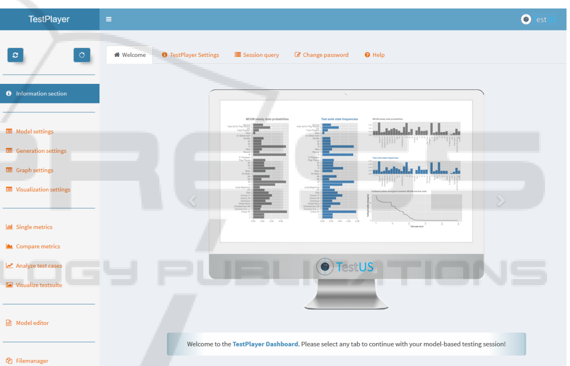

Figure 5: The TestPlayer Dashboard.

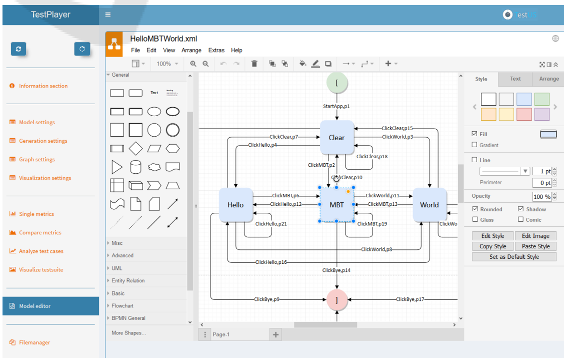

Figure 6: The graphical TestPlayer model editor.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

2.1 Markov Chain Usage Model for a Simple Web GUI Application

Usage models can be created in the `Model editor` section of the TestPlayer Dashboard by means of a graphical editor that is based on draw.io[5] (Fig. 6).

In the following, we will briefly present the web application `HelloMBTWorld`[6] (Fig. 7), which serves as a running example to explain the basic approach.

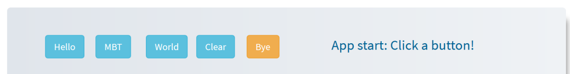

Figure 7: Web application `HelloMBTWorld`.

Diagrams (Fig. 8 - Fig. 11) show how `HelloMBTWorld` behaves when the four buttons `Hello`, `MBT`, `World` and `Clear` are pressed. The red `Bye` button terminates the execution of the application.

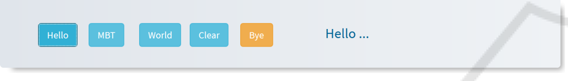

Figure 8: Pressing the `Hello` button.

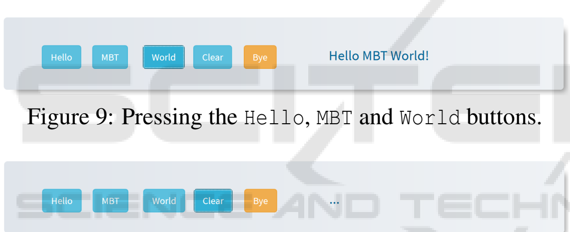

Figure 9: Pressing the `Hello`, `MBT` and `World` buttons.

Figure 10: Pressing the `Clear` button.

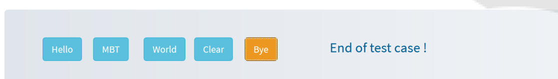

Figure 11: Pressing the `Bye` button.

The corresponding usage model created by the test engineer using the TestPlayer model editor is shown in Figure 12.

After starting the web application, usage state `Clear` is directly reached from the *start state* `[` and then, depending on whether the click events `ClickHello`, `ClickMBT`, `ClickWorld` or `ClickClear` are selected the usage states `Hello`, `MBT`, `World` or `Clear` are reached. Pressing the `Bye` button generates the click event `ClickBye` and terminates the test case when the *final state* `]` is reached.

The additional *generic* probabilities `p1` up to `p21` at the edges behind the input events define the transition probabilities for selecting the respective input event `e1` up to `e21`. Prior to the generation of the test cases, the TestPlayer replaces the generic values by *concrete* probability values that either originate from a given test profile or are calculated based on a *uniform geometric* distribution.

[5]https://about.draw.io/

[6]https://testus.eu/HelloMBTWorld/

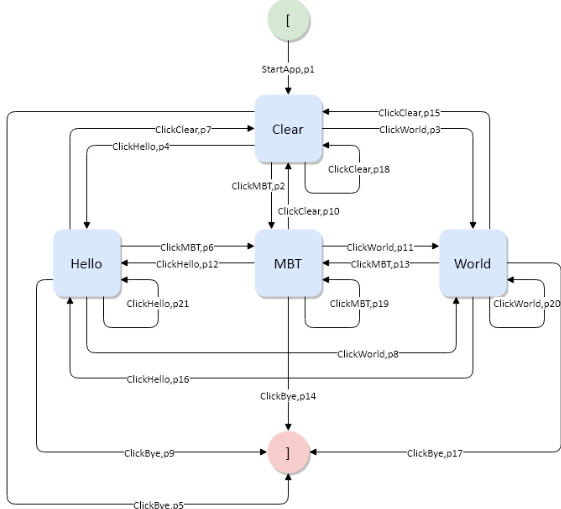

Figure 12: MCUM for the web application `HelloMBTWorld`.

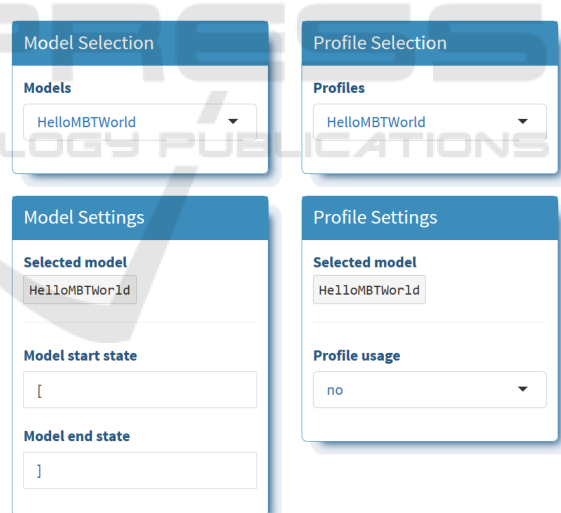

Figure 13: TestPlayer model settings.

In section `Model settings` the essential parameters for the automatic generation of test cases using the TestPlayer can be specified (Fig. 13). These include

- `Models`: file name of the usage model
- `Model start state`: name of the start state of the usage model
- `Model end state`: name of the final state of the usage model

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

- `Profile usage`: declaration, whether a statistical usage profile (yes) or a uniform distribution (no) will be employed for the generation algorithm of the test cases
- `Profiles`: file name of the statistical usage profile if any.

2.2 Automatic Generation of Test Cases

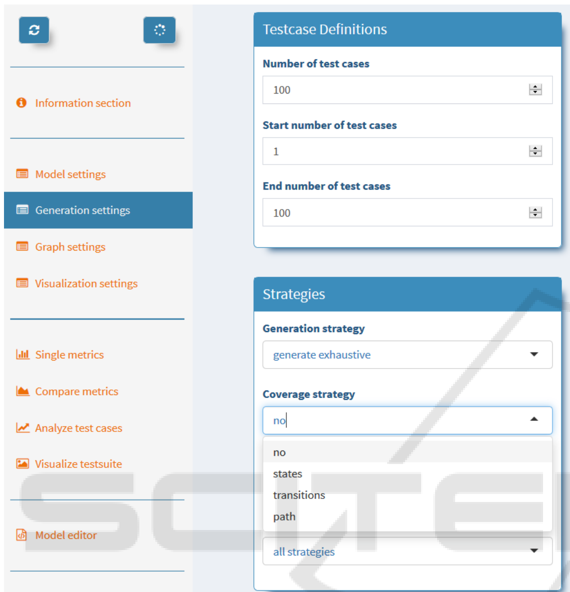

Figure 14: TestPlayer test suite generation settings.

By applying the given usage model, the TestPlayer Dashboard offers simple user interactions (Fig. 14) to automatically generate dedicated test suites that have specific characteristics, i.e.

- complete *coverage* of all *usage states*
- *coverage* of all possible *transitions* between the usage states
- *coverage* of all *loop-free paths*, i.e. no state transition is selected twice within a test case.

The default number of test cases that are generated by the TestPlayer is 100 but can easily be changed within the `Testcase definitions` (Fig. 14). Test suites that possess the specific characteristics defined above arise by reduction with respect to the given coverage and sort criteria. In this way, test suites with different properties (Dulz, 2013) can be created automatically.

Figure 15 shows a single test case from a test suite consisting of four test cases, which achieves a complete coverage of all usage states for the usage model in Figure 12. The test suite was generated using the

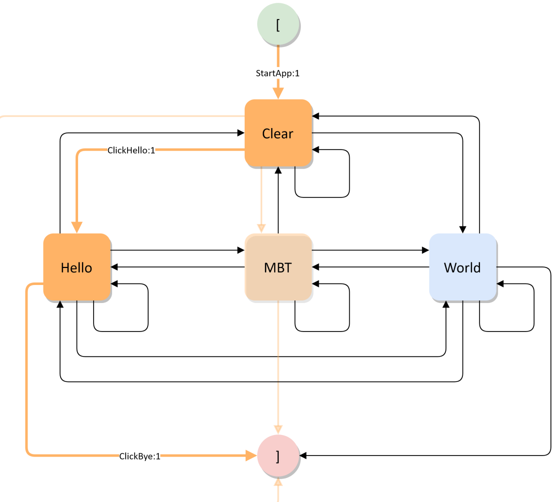

Figure 15: Test case visualization for the MCUM shown in Figure 12.

sort criterion `length`, i.e. the four test cases were selected from 100 statistically generated test cases after all test cases were sorted according to their length.

Test cases are visualized by highlighting the relevant states and transitions (bold orange coloring) and show the already achieved coverage of usage states or state transitions (represented by a light orange coloring). The number behind the colon of the click events indicates how often the specified state transition is performed during the execution of the test case. In the case of longer loops during the execution of test cases, a single transition can be traversed several times.

In addition to the graphical representations, the TestPlayer provides a textual description of the generated test suite. Textual variants of the test suite are intended for documentation purposes as well as to export test cases for the test execution using a JSON-like notation.

The *JSON-like* test suite description is as follows:

- a *test suite* T consists of test cases $TC_1 \cdots TC_m$ notated as $[[TC_1],[TC_2],\cdots[TC_m]]$
- a *test case* T consists of test steps $TS_1 \cdots TS_n$ notated as $[[TS_1],[TS_2],\cdots TS_n]]$
- a *test step* TS consists of usage states US_{from} and US_{to} and the transition event E notated as $[US_{from},E,US_{to}]$

A concrete test step example for the test case visualization in Fig. 15 looks as follows:

```
["Clear","ClickHello","Hello"]
```

The complete test case visualized in Fig. 15 in the JSON-like description is as follows:

```
[
 ["[", "StartApp", "Clear"],
```

```
    ["Clear", "ClickHello", "Hello"],
    ["Hello", "ClickBye", "]"]
  ]
```

How to use a test suite in the JSON-like representation for an automated test execution process is discussed in section *Eclipse for the Automated Test Suite Execution*.

2.3 Graphical Representation of Characteristic Test Suite Properties

Once a test-suite has been generated, specific metrics can be used to graphically analyze and evaluate its properties and to assess the quality of the test suite.

Metric *SSP* compares the probability distribution of usage states in statistical equilibrium for the usage model and the relative frequencies of the corresponding usage states in the generated test suite. As can be seen in Fig. 16, the theoretical probability values for the individual usage conditions of the MCUM are well mapped in the test suite.

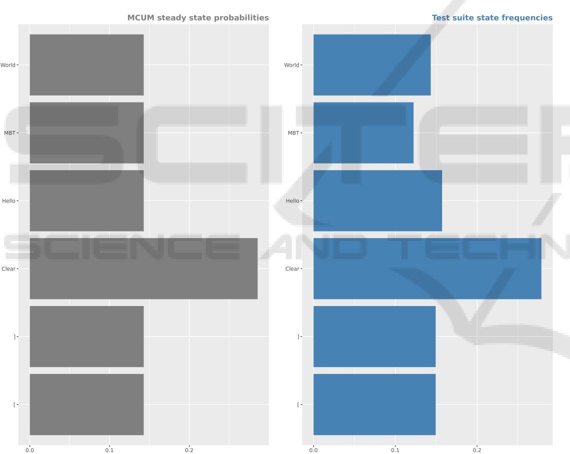

Figure 16: Steady state probabilities of the MCUM vs. relative frequencies of the test suite.

In addition, the TestPlayer offers further metrics, which are discussed in more detail in (Dulz, 2013):

- *SSV*: comparison of the average number of test cases that are necessary to visit a usage state once in the usage model and during the test execution
- *KL*: visualization of the Kullback/Leibler divergence and the mean weighted deviation (Dulz, 2011) between the usage model and the test suite
- *SSP.N*, *SSV.N*, *KL.N*: corresponding metrics for test suites that cover all nodes of the usage model.

3 ECLIPSE FOR THE AUTOMATED TEST SUITE EXECUTION

Eclipse is an open source programming environment for modeling and developing all kinds of (application) software, which fits ideally into the comprehensive test framework. There exist plug-ins for all common programming approaches, e. g.

- *Java*: applications, client/server side programming, Android, ···
- *PHP*: server side programming
- *JavaScript/CSS/HTML5*: wep applications
- *JUnit*: white box unit tests of Java components
- *Selenium*: software-testing framework for web applications

```
// ...
String browser=null;
// select browser type from command line
browser=args[0];
if (browser.equalsIgnoreCase("Firefox")) {
  // browser path
  String pathToGeckoDriver="geckodriver";
  System.setProperty("webdriver.gecko.driver",
                     pathToGeckoDriver);
  driver=new FirefoxDriver();
} else if (browser.equalsIgnoreCase("Chrome")) {
  // browser path
  String pathToChromeDriver="chromedriver";
  System.setProperty("webdriver.chrome.driver",
                     pathToChromeDriver);
  driver=new ChromeDriver();
} else {
  throw new Exception("Browser not defined!");
}
// ...
// start app from URL
public static void startApp(String URL){
  driver.get(URL);
}
// find HTML element by ID
public static void byID(String ID) {
  driver.findElement(By.id(ID)).click();
}
// find HTML element by HTML tag
public static void byTag(String tag) {
  driver.findElement(By.tagName(tag)).click();
}
// ...
```

Figure 17: Elements of the Selenium web driver Java API for Eclipse.

Automated testing of web applications requires additional drivers to provide the ability to automatically access the respective web browser. We use the

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
for (String key : keyClicks) {
  switch (key) {
    case "ClickHello":
      byID("Hello");
      Thread.sleep(time);
    break;
   case "ClickMBT":
      byID("MBT");
      Thread.sleep(time);
    break;
    // ...
  }
}
```

Figure 18: Main Java `switch()` for executing a single test step.

test automation framework Selenium[7], which can be easily integrated into an Eclipse-based test environment and offers a common Java API for the main web browsers (Fig. 17).

For web applications to be tested automatically, a testing interface must be provided that simulates the state-based logic of the Markov chain. For this purpose, each test step describes a state transition that implements the desired test request. The Java `switch()` statement in Fig. 18 implements a typical programming pattern that is performed during the execution of a given test suite. `String keyclicks` provides single transition events `key` that trigger the test step. To control the duration of the corresponding display action, the time-controlled method `Thread.sleep(time)` is used in addition. The IDs that are used as input parameters for method `byID()` are the corresponding HTML `id` attributes in the `index.html` file of the web application, e.g.

```
<button id="Hello" type="button"
        class="btn btn-info">Hello
</button>
```

Before testing a web application, the test engineer must first select the type of the web browser to start the correct driver. Concrete values for the driver are `FirefoxDriver()` for the Mozilla web browser and `ChromeDriver()` for the Google web browser. The web application can then be started via the `startApp(String URL)` method for the given URL and subsequently automatically tested using the previously generated test cases, as shown in Fig. 17.

During the test, specific methods from the Selenium API are used to navigate inside the web application, such as `byID(String ID)` for clicking an HTML element with the given identifier `ID` or `byTag(String tag)` for clicking the next HTML element with the given HTML tag.

[7]https://www.seleniumhq.org/

```
[
  [
    ["[", "StartApp", "Clear"],
    ["Clear", "ClickBye", "]"]
  ],
  [
    ["[", "StartApp", "Clear"],
    ["Clear", "ClickMBT", "MBT"],
    ["MBT", "ClickBye", "]"]
  ],
  [
    ["[", "StartApp", "Clear"],
    ["Clear", "ClickHello", "Hello"],
    ["Hello", "ClickBye", "]"]
  ],
  [
    ["[", "StartApp", "Clear"],
    ["Clear", "ClickWorld", "World"],
    ["World", "ClickBye", "]"]
  ],
  {}
]
```

Figure 19: Test suite that covers all states of the web application HelloMBTWeb in Fig. 12.

When testing the `HelloMBTWorld` web application by means of the test suite given in Fig. 19, intermediate results are logged in the Eclipse Console window showing which transition event has been executed by each test case:

```
TestCase[1]=<StartApp ClickBye>
TestCase[2]=<StartApp ClickMBT ClickBye>
TestCase[3]=<StartApp ClickHello ClickBye>
TestCase[4]=<StartApp ClickWorld ClickBye>
```

At the end of the test, a window pops up displaying the test result:

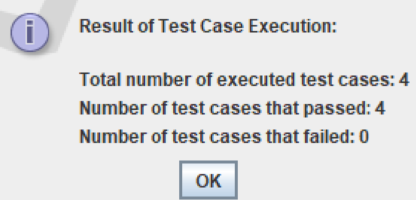

Figure 20.

4 MODEL-BASED TESTING OF MULTILINGUAL WEBSITES

So far, we have shown how simple web applications can be tested by means of model-based testing. Now we are going a step further and focus on testing multilingual websites. For this purpose, a suitable usage model must first be created. In the following, we will

explain how the TestUS[8] website can be tested.

4.1 Language-dependent Usage Models

The corresponding usage model of the website is given in Fig. 21.

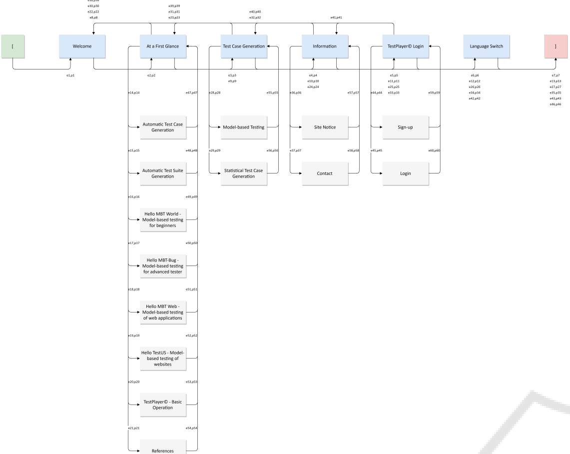

Figure 21: Usage model of the TestUS website.

The TestUS website always starts in the usage state `Welcome`. From there, you can reach the main usage states `At a First Glance`, `Test Case Generation`, `Information`, `TestPlayer Login` and `Language Switch`. The main usage states correspond to the selection menus in the top menu bar of the TestUS website. From the main usage states, you get to the other usage states of the website, e.g. `At a First Glance` provides access to

- `References`
- `Automatic Test Case Generation`
- `Automatic Test Suite Generation`

...

`TestPlayer Login` offers the access to

- `Sign-Up` and
- `Login`

and `Language Switch` switches the representation language of the website between English and German.

A typical test case generated automatically by the TestPlayer looks like the one shown in Fig. 22.

After changing the language from English to German by clicking on the language switch of the website a different usage model must be used to create correct test cases in German. The German test case (Fig. 22) now looks like the one in Fig. 23.

[8]https://testus.eu

```
[
  ["[","e1","Welcome"],
  ["Welcome","e5","TestPlayer Login"],
  ["TestPlayer Login","e41","Information"],
  ["Information","e37","Contact"],
  ["Contact","e58","Information"],
  ["Information","e30","Welcome"],
  ["Welcome","e6","Language Switch"],
  ["Language Switch","e46","]"]
]
```

Figure 22: Test case for testing the English version of the TestUS website.

```
[
  [","e1","Willkommen"],
  ["Willkommen","e5","TestPlayer Login"],
  ["TestPlayer Login","e41","Informationen"],
  ["Informationen","e37","Kontakt"],
  ["Kontakt","e58"," Informationen"],
  ["Informationen","e30", "Willkommen"],
  ["Willkommen","e6","Sprachschalter"],
  ["Sprachschalter","e46","]
]
```

Figure 23: Test case for testing the German version of the TestUS website.

The main differences between the two usage models are the different names of the usage states in the selected language. The structure and the generic transition events are not affected by this change.

4.2 Generic Usage Model

For that reason, it makes sense to provide a usage model containing *generic state names* that can be mapped to concrete names of the respective languages during the test execution. This task can be performed by the TestPlayer that can add generic state names Ni $(i = 1, \cdots)$ to an incomplete usage model. The result is a generic usage model having 20 generic usage states $N1$ to $N20$ and 60 generic transition events $e1$ to $e60$.

The concrete English test case in Fig. 22 now looks like the following generic one in Fig. 24.

```
[
  ["[","e1","N1"],
  ["N1","e5","N5"],
  ["N5","e41","N4"],
  ["N4","e37","N20"],
  ["N20","e58"," N4"],
  ["N4","e30", "N1"],
  ["N1","e6","N6"],
  ["N6","e46","]"]
]
```

Figure 24: Generic test case for multilingual testing of the TestUS website.

SCIENCE AND TECHNOLOGY PUBLICATIONS

To test websites successfully with the presented techniques, individual HTML elements must be labeled with unique identifiers. Therefore, the HTML source code of the English TestUS website contains HTML markups for the top menu bar as shown in Fig. 25.

```
<ul class="navbar-nav">
  <li id="N1_en">Welcome</li>
  <li id="N2_en">At a First Glance
<!-- ... -->
  <li id="N3_en">Testing Methods</li>
<!-- ... -->
  <li id="N4_en">Information
  <ul>
    <li id="N17_en">Site Notice</li>
    <li id="N18_en">Contact</li>
  </ul></li>
  <li id="N5_en">TestPlayer© Login
  <ul>
    <li id="N19_en">Sign-up</li>
    <li id="N20_en">Login></li>
  </ul></li>
  <li><img id="N6_en" src="./img/de.gif"
           alt="Switch to German">
  </li>
</ul >
```

Figure 25: HTML markups for the English top menu bar.

As easily seen, the identifiers are composed of the generic state names and a label for the language that is used. The corresponding German description looks as shown in Fig. 26.

```
<ul class="navbar-nav">
  <li id="N1_de">Willkommen</li>
  <li id="N2_de">Auf den ersten Blick
<!-- ... -->
  <li id="N3_de">Testfallgenerierung</li>
<!-- ... -->
  <li id="N4_de">Informationen
  <ul>
    <li id="N17_de">Impressum</li>
    <li id="N18_de">Kontakt</li>
  </ul></li>
  <li id="N5_de">TestPlayer© Login
  <ul>
    <li id="N19_de"> Registrierung</li>
    <li id="N20_de">Login></li>
  </ul></li>
  <li><img id="N6_de" src="./img/en.gif"
           alt=" Switch to English">
  </li>
</ul >
```

Figure 26: HTML markups for the German top menu bar.

During the test execution, the system must switch to the other language when the language switch is detected inside a test case. The Java code in Fig. 27 shows how Eclipse is performing this task.

For websites to be tested automatically, a testing interface must be provided that simulates the state-based logic of the usage model. In analogy to Fig. 18, we use a Java `switch()` statement consisting of 60+1 entries for the generic transition events and an additional default entry to react when an invalid input occurs.

```
// set web site language
public static void switch_Language() {
  if (webSiteLanguage.equals("en")) {
  // switch to other language
  webSiteLanguage = "de";
  N1 = "N1_de";
  N2 = "N2_de";
  // ...
  N20 = "N20_de";
  } else if (webSiteLanguage.equals("de")) {
  // switch to other language
  webSiteLanguage = "en";
  N1 = "N1_en";
  N2 = "N2_en";
  // ...
  N20 = "N20_en";
  }
}
```

Figure 27: Java code for switching the language during the test.

```
case "e45":
  byID(N20);
  // scroll part
  if (scrollMode == 1) {
    Thread.sleep(mainTime);
    for (int second = 0;; second++) {
      if (second >= 30) {
        break;
      }
      jsExecutor.executeScript("window.
                               scrollBy(0,50)");
      Thread.sleep(100);
    }
    jsExecutor.executeScript("window.
                             scrollTo(0,0)");
  } else
  Thread.sleep(mainTime);
break;
```

Figure 28: Java code for testing the TestUS `Login` page.

The Eclipse `Run Configuration` can be set via the parameter `s` (scroll mode) to indicate whether the individual pages should be scrolled during testing of the website. This feature is used for controlling the run-time of the test execution. In case of a lengthy test suite, e. g. for a desired transition coverage, the scroll mode can be switched off to get a quick overview of the behaviour of the website. When the scroll mode is activated the duration of the display and the scroll action are time-controlled via the class attribute `mainTime` and the sleep method `Thread.sleep()`. The Java code fragment in Fig. 28 shows the actions that are triggered when selecting the TestUS `Login` page within a test case, which is indicated by the generic state name *N*20.

When the generic transition event *e*45 appears in a test case the `TestPlayer Login` item is clicked automatically in the selected web driver by performing the method `byID(N20)`. After a predefined timeout of `mainTime`, which controls the web page display time, the JavaScript engine of the web browser must ex-

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

ecute the JavaScript code `window.scrollBy(0,50)` to scroll down by 50 pixels. When the bottom part of the web page is reached, the browser scrolls automatically to the top of the web page by executing the JavaScript code `window.scrollTo(0,0)`.

Test suites can also be animated during the test execution, as seen in Fig. 29. An interactive video showing the test execution in more detail can be downloaded from the TestUS website[9].

In this example, the test suite covers all states of the usage model in Fig. 21 and consists of a total of 19 test cases. Test cases are sorted by length, i.e. shorter test cases are executed first. In addition, the highlighting strategy of the test cases is set to accumulated, which is explained in more detail in the next subsection.

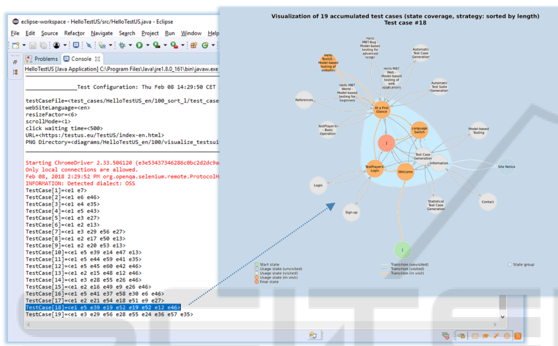

Figure 29: Animated test execution of the TestUS website.

4.3 Graphical Visualization of Test Cases and Test Suites

In the visualization section the TestPlayer provides a variety of options to visualize individual test cases or groups or a complete test suite (Fig. 30).

The `Visualization strategy` determines whether test cases are labeled as a group (accumulated) or if each test case (single test case) is labeled individually. This means in detail

- *single test case:* labeling of nodes and edges is carried out independently and individually for all test cases
- *accumulated:* usage states (nodes) that are already labeled in the previous test cases keep their labels; in addition, new labels are added for those nodes that appear for the first time in a test case.

The Visualization layout determines which graphic layout for representing individual test cases of a given test suite shall be chosen. Below different

[9]https://testus.eu/TestUS/video/HelloTestUS_en_state_coverage_scroll.mp4

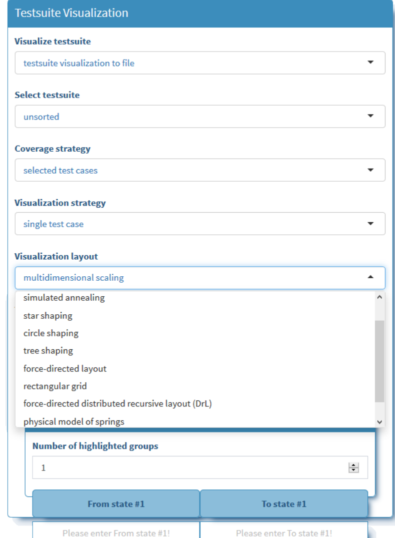

Figure 30: Visualization setting in the TestPlayer dashboard.

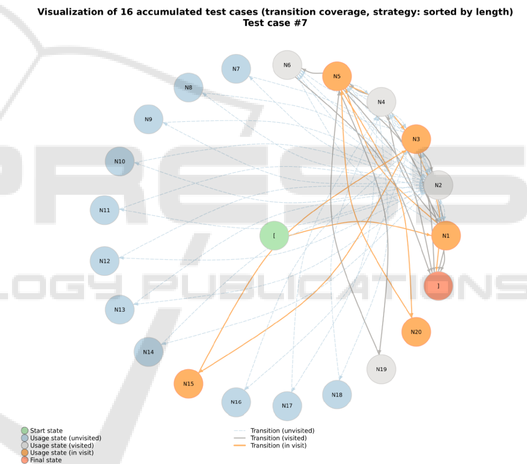

Figure 31: Layout strategy *star shaping*.

layouts are shown for the same test case to compare the differences of the visualization layouts. Possible layout strategies are

- *annealing:* place vertices of a graph on the plane, according to simulated annealing
- *multidimensional scaling:* place points from a higher dimensional space in a two-dimensional plane, so that the distance between the points are kept as much as this is possible
- *star shaping:* places one vertex in the center of a circle and the rest of the vertices equidistantly on the perimeter (Fig. 31)
- *circle shaping:* place vertices on a circle, in the order of their vertex ids

- *tree shaping:* a tree-like layout, which is perfect for trees and acceptable for graphs with not too many cycles (Fig. 32)

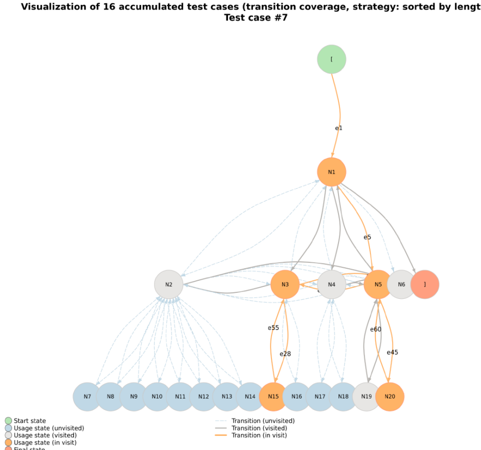

Figure 32: Layout strategy *tree shaping*.

- *force-directed layout:* layout algorithm, that scales relatively well to large graphs
- *rectangular grid:* places vertices on a two-dimensional rectangular grid (Fig. 33)

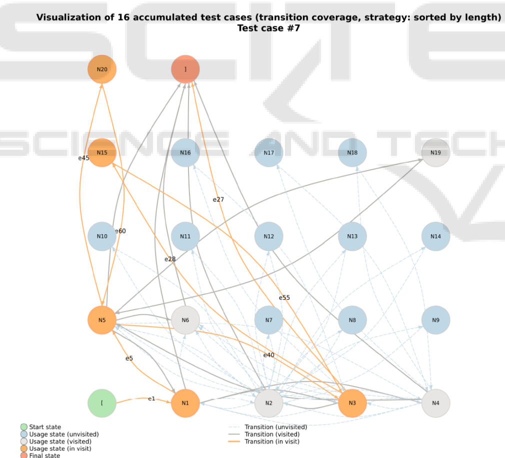

Figure 33: Layout strategy *grid shaping*.

- *force-directed distributed recursive layout*
- *physical model of springs:* place the vertices on the plane based on a physical model of springs
- *uniformly random:* randomly places vertices on a [-1,1] two-dimensional square

4.4 Test Focusing by Means of Adapted Usage Profiles

Of special importance for the validation of the SUT are customer-specific usage profiles that focus the test execution on selected usage states or sets of usage states. This is achieved by

- avoiding a transition (Si, Sj) that is starting in usage state Si and ending in usage state Sj by setting the corresponding probability value $p(Si, Sj) = 0$
- forcing a transition (Si, Sj) by setting the corresponding probability value $p(Si, Sj) = 1$.

The result is an adapted usage profile that is used to generate the test-suite. In this way, you can easily describe various user classes that visit the website in different ways.

Fig. 34 shows an adapted test suite that focuses only on those visitors of the TestUS website (Fig. 21) who access the top menu `At a First Glance`. The corresponding usage profile is as follows:

```
p2=1, p8=0, p9=0, p10=0, p11=0, p12=0
```

A test case, which *must* be performed during the test procedure due to special safety requirements, is often referred to as the *happy path*. The implementation of a happy path can also be easily realized with the concept of adapted usage profiles.

5 CONCLUSION AND FINAL REMARKS

This paper discusses techniques that make it possible to prepare and perform model-based testing of web applications and multilingual websites using a versatile tool chain consisting of the TestUS TestPlayer and the modeling framework Eclipse.

For this purpose, the simple web application `HelloMBTWorld` is used to explain the essential steps

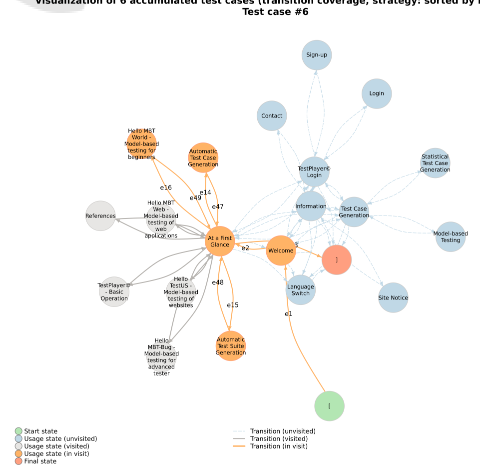

Figure 34: Visualization of an adapted test suite focusing on usage state `At a First Glance` of the usage model in Fig. 21.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

for a model-based test process that applies statistical usage models to generate and visualize appropriate test suites automatically.

Using the TestPlayer, it is easy to perform a tool-driven assessment of the generated test suites. By means of the provided diagrams a test engineer can decide very quickly which and how many test cases are needed to accomplish a certain test objective.

The *key insights* from our projects in recent years and this paper can be summarised as follows:

- Model-based techniques that use graphical representations of usage models can help even inexperienced test engineers prepare and perform their tests.
- Graphical usage models facilitate the setting of the test focus on those areas of the SUT that need to be tested.
- Generic usage models, which can be adapted to a given language environment during the test execution, allow the testing of multilingual websites.
- Adapted profiles support the selective generation of test suites. Based on adopted profiles different user groups that interact with the SUT can be distinguished by different test suites that are used during the test execution. How to systematically derive an adopted profile is explained in more details in (Dulz et al., 2010).
- The Eclipse modeling framework in combination with the TestPlayer tool chain provides a versatile tool environment for model-based testing of web applications and websites.

REFERENCES

Dulz, W. (2011). A Comfortable TestPlayer for Analyzing Statistical Usage Testing Strategies. In *ICSE Workshop on Automation of Software Test (AST '11)*, Honolulu, Hawaii.

Dulz, W. (2013). Model-based strategies for reducing the complexity of statistically generated test suites. In *SWQD 2013, Vienna, Austria, January 15-17, 2013. Proceedings*, pages 89–103.

Dulz, W., Holpp, S., and German, R. (2010). A Polyhedron Approach to Calculate Probability Distributions for Markov Chain Usage Models. *Electronic Notes in Theoretical Computer Science*, 264(3):19–35.

Dulz, W. and Zhen, F. (2003). MaTeLo - Statistical Usage Testing by Annotated Sequence Diagrams, Markov Chains and TTCN-3. In *IEEE International Conference on Quality Software (QSIC 2003)*, pages 336–342.

El-Far, I. K. and Whittaker, J. A. (2001). Model-based Software Testing. In Marciniak, J., editor, *Encyclopedia on Software Engineering*. Wiley.

Gutjahr, W. (1997). Importance sampling of test cases in markovian software usage models. *Probability in the Engineering and Information Sciences*, 11:1936.

M. Utting, B. L. (2007). *Practical Model-Based Testing*. Elsevier.

Musa, J. D. (1996). The operational profile. *NATO ASI Series F, Computer and system sciences*, 154:333–344.

Poore, J., Walton, G., and Whittaker, J. (2000). A constraint-based approach to the representation of software usage models. *Information & Software Technology*, 42(12):825–833.

Prowell, S. (2000). Computations for Markov Chain Usage Models. Technical report, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, USA. UT-CS-03-505.

Prowell, S. J. (2003). Jumbl: A tool for model-based statistical testing. In *HICSS*, page 337.

Rosaria, S. and Robinson, H. (2000). Applying models in your testing process. *Information and Software Technology*, 42:815–824.

Sayre, K. and Poore, J. (2000). Stopping criteria for statistical testing. *Information and Software Technology*, 42(12):851857.

Takagi, T. and Furukawa, Z. (2004). Constructing a Usage Model for Statistical Testing with Source Code Generation Methods. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC04)*.

Tian, J. (2005). *Software Quality Engineering*. John Wiley&Sons.

Walton, G. and Poore, J. (2000). Generating transition probabilities to support model-based software testing. *Software Practice and Experience*, 30(10):1095–1106.

Walton, G. H., Poore, J. H., and Trammell, C. J. (1995). Statistical Testing of Software Based on a Usage Model. *Software - Practice and Experience*, 25(1):97–108.

Whittaker, J. A. and Poore, J. H. (1993). Markov Analysis of Software Specifications. *ACM Transactions on Software Engineering and Methodology*, 2(1):93–106.