WHITEPAPER

THE TESTUS TESTPLAYER©

A Versatile Tool Environment for Model-Based Testing

Model-based testing is an innovative test approach to improve the effectiveness and efficiency of the testing process. A model-based tester uses models to control test design and analysis and takes advantage of models for other test activities such as test case generation and test report generation. *(ISTQB® MBT certification)*

The Model-based Testing Company

TestUS

THE TESTUS TESTPLAYER©

A Versatile Tool Environment for Model-Based Testing

Table of Contents

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.

Model-based Testing

One method in which the test cases are generated from a model is called Model-based Testing (*MBT*). The relationship between a model that describes those parts of a given SUT (System under test) that need to be tested to generate (automatically) test cases derived from the (graphical) model is illustrated in Fig. 1.

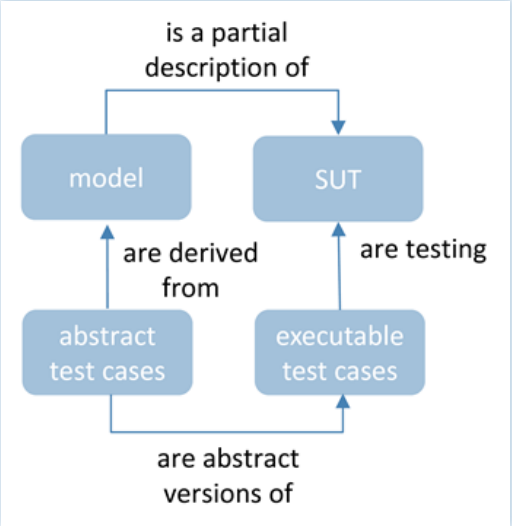


Fig. 1. General approach of Model-based Testing

In general [4], 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.

Test cases, which are generated from a system specification 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.

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 of statistical usage models, also called Markov chain usage models (*MCUM*), individual test cases or complete test suites can (automatically) be derived by traversing the MCUM. Statistical usage models are graphical representations to define all possible usage steps and scenarios on a given level of abstraction as shown in Fig. 2.

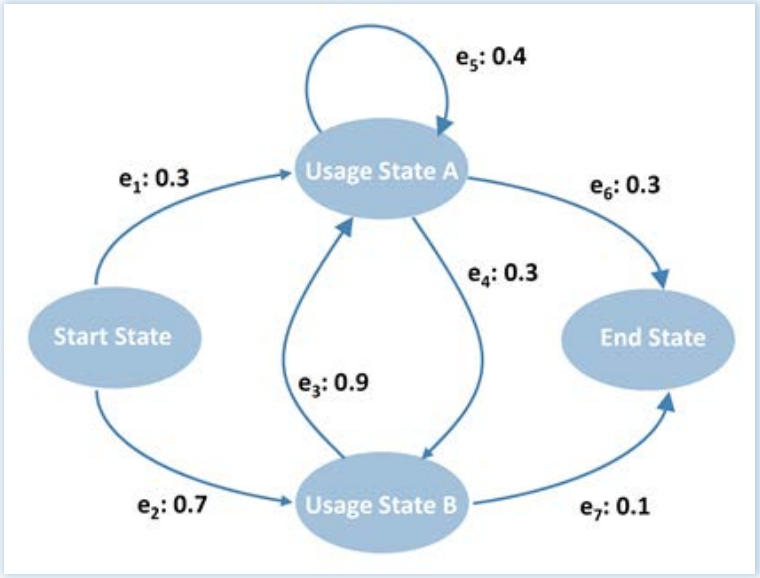


Fig. 2. Statistical usage model for modeling the usage behavior of system users

Statistical usage models consist of

- 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.

The probability that a user interaction triggers an event e_j is called transition probability and is given behind a colon, e.g. `e4:0.3` to change from `Usage State` A into `Usage State` B (Fig. 2). By adjusting the probability values of the usage distribution, i.e. the operational usage profile, 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 usage model 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.

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 usage model (steady state analysis)
- probability for the occurrence of a certain state/transition in each 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.

Our expertise and experience from various national and international projects, such as reviewing the recent ISTQB Foundation Level Model-Based Tester Syllabus, show that statistical usage 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.

(References [6], [9], [10], [11])

Automated Statistical Usage Testing

Given the promising properties and results of statistical usage models, there is a demand to provide a suitable tool environment for auto-mating the test case generation and test execution process.

Using the TestPlayer©, it is possible to assess the quality of the generated test cases based on graphical representations at an early stage and even before the actual test execution.

The TestPlayer© allows the characteristic properties of a test suit to be seen very quickly, such as

- the *maximum length of a test case*, i.e. the number of single test steps between the start and the end state of the usage model,
- the *mean length of the test cases* in a test suite,
- the *accordance* between the *usage profile* of the usage model and *the usage frequencies* of the generated test suite,
- the *coverage* of all *usage states* after having executed the test suite and
- the *coverage* of all state *transitions* after having executed the test suite.

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 test objective. As illustrated in Fig. 3, 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.

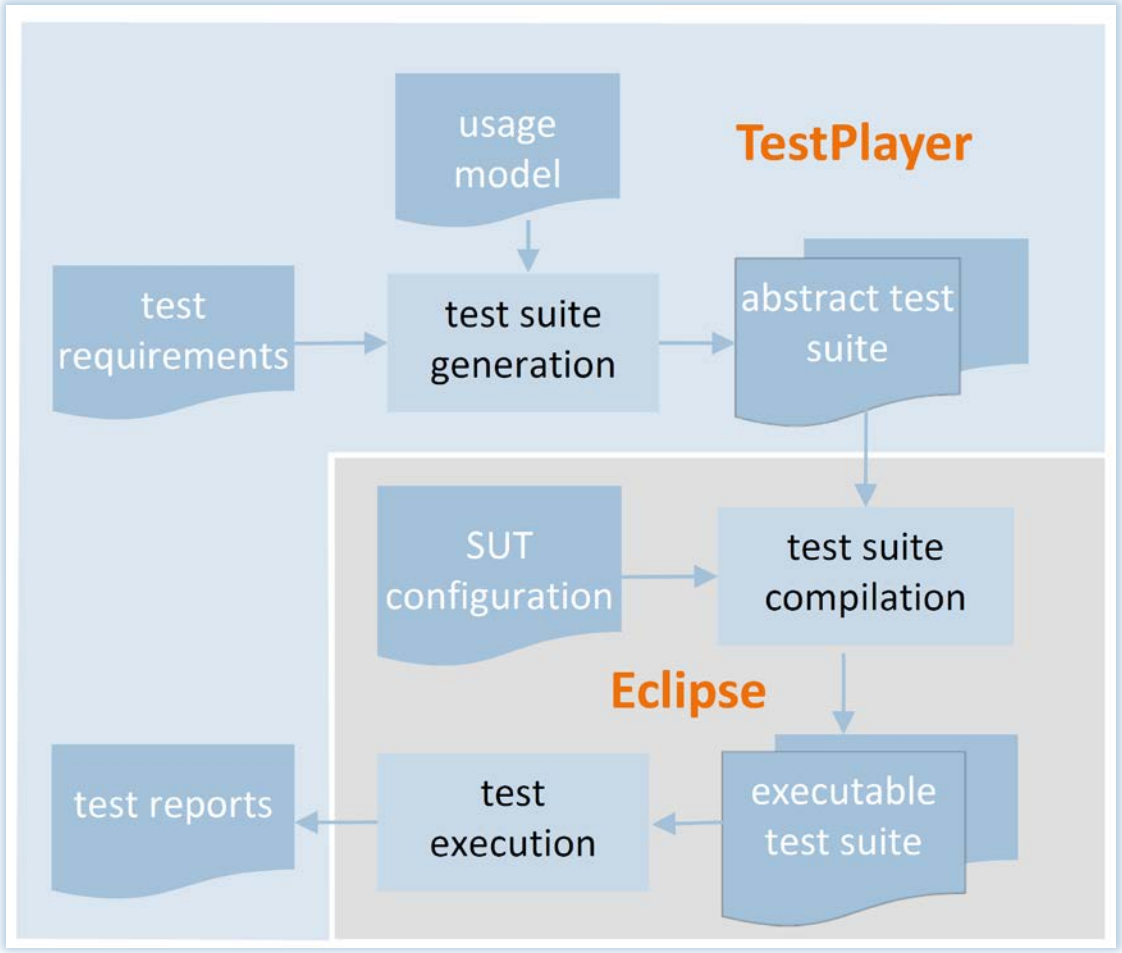


Fig. 3. A versatile tool environment for statistical usage testing

Handling the Test Requirements

Although requirements engineering is usually not considered as part of the test process, it is still crucial for the subsequent test phases. There are some points that need to be considered at this early stage to ensure the success of test activities, especially if statistical usage tests are to be applied.

The ability for testing consists of two aspects

- Guarantee that the input documents for the test process have the required quality level.
- Ensuring that the system under test meets all technical requirements for test execution.

Roles
Requirements Engineer
Domain Specialist
Test Manager
Test Engineer

Input
User Requirements

Output
Test Requirements
Testability Analysis Report

Practical Tips

- Involve test experts in the early review of requirements and design specifications to avoid testability issues later.
- The precondition for MBT is that the requirements are sufficiently detailed and precise to derive a formal model.
- Try using activity diagrams, state diagrams, usage scenarios, etc. instead of plain text.
- Make sure that you plan interfaces that can later be used by test automation tools for accessing the SUT.

Creation of the Usage Model

Compared to a conventional test project, in which test cases are prepared in a tabular format, for example, a model-based test project requires the development of a test model.
This model represents the usage behavior from a test perspective and contains all scenarios, constraints and dependencies that are relevant for testing.
Modeling can involve considerable effort, especially if the input documents do not have the required level of detail and frequent requests are required.
On the other hand, this is an excellent way to identify errors and inconsistencies at an early stage before the implementation phase.
Thus, this initial investment helps to avoid serious problems in later phases of the project and to significantly improve the maintainability of the tests.
However, it should be checked in advance when the ROI will be achieved and whether MBT can be considered for a specific test order.

Roles
Requirements Engineer
Test Engineer
Software Designer

Input
Test Requirements

Output
Usage Model

Practical Tips
- Avoid the reuse of a model that has also been used for code generation. If both the code and the tests are generated from the same model, no deviations will be detected! Instead, try to create a specific model from the test perspective.
- Creating a test model is a challenging task that requires a variety of skills (test design, abstraction, under-standing requirements, technical background). Don't expect end users to accomplish this task, employ specialists instead!
- Start modeling on an abstract level and then add more details step by step.
- Make sure that requirements engineers and software designers are available to answer any questions the test developers may have.
- Make sure that the model components are reusable and maintainable, e.g. use parameterizable building blocks.

Generation of Abstract Test Suites

The TestPlayer can be used to automatically create abstract test suites based on the provided usage model.
Different generation strategies and coverage criteria can be selected depending on the predefined test requirements.
In conjunction with the test suites, the TestPlayer creates statistics, graphical analyses concerning the quality of the generated test suites and visualizations of the generated test cases.

Roles
Test Engineer

Input
Usage Model

Output
Abstract Test Suites
Statistics
Graphics
Visualizations

Practical Tips
- Select the test coverage based on the model complexity and the relevance of the associated requirements.
- Combine different test strategies, e.g. sort test cases by length of by frequency, to increase the detection potential of errors.
- The goal of the integration is to transfer the generated abstract test cases directly to the test execution tool, i.e Eclipse.

Building an Executable Test Suite

To run the test automatically, the abstract test suite must be linked to a test execution script that contains concrete instructions how to interpret the events in a certain usage state.
One way to do this is to implement an adapter or script for each event used in the statistical usage model. However, this requires considerable effort and advanced programming knowledge.
A much more efficient alternative is to use an MBT tool such as Eclipse, which can import GUI information and automatically generate executable scripts. In this case, each event in the usage model only needs to be mapped to the affected GUI element.
According to the requirements it must be decided whether test data can be imported from an existing source, created manually or generated automatically.

Roles
Test Engineer

Input
Abstract Test Suite
SUT Configuration

Output
Executable Test Suite

Practical Tips
- Simple test data design methods such as equivalence partitioning and limit analysis should only be used for data fields that have no dependencies on others.
- In all other cases, methods such as cause/effect analysis should be used.

Execution of the Test

Some projects use a test generator to create manual test instructions. More frequently, a testing tool performs the execution of the test automatically.
In both cases, a suitable interface must be provided to send test instructions to the SUT and to check the results.

Roles
Test Engineer

Input
Executable Test Suite

Output
Test Verdicts
Test Logs

Practical Tips

- Do not expect that the test automation will work immediately - rather, it is an iterative process. Ensure that the generated test cases work as expected (GUI object recognition, timing, etc.). If necessary, correct the model until the tests run smoothly.
- Test scripts derived from a model typically require the system under test to be in a well-defined state and behave predictably at the beginning of each test.
  For example, changes in the database or unexpected pop-up messages can cause the scripts to fail if they are not considered in the model.
- In the case of deviations from the expected behavior, check whether the problem was caused by an error in the software or by an error in the test model.

Generation of Test Reports

The test logs created during the test process must be converted into a comprehensive test report that informs the responsible project manager about the current project status.

Roles
Test Engineer
Test Manager

Input
Test Logs
Test Verdicts

Output
Test Reports

Practical Tips

- For statistical usage testing, it is important that statistics on the model coverage are included, such as the percentage of states, transitions, or scenarios that are covered by the test suite.
- Many test management tools can generate reports, provided they have access to all the necessary data. In this context, a good integration with other testing tools is also important.

Maintenance of the Statistical Usage

For each new version to be tested, the tests must be adapted and repeated.
The manual update of each test script is not practical because it can be very time-consuming.
On the other hand, updating a model that consists of reusable components is far less time-consuming, since only a few parts need to be changed before perfectly updated test scripts are created.

Roles
Test Engineer
Test Manger

Input
Statistical Usage Model
Change Requests

Output
Updated Usage Model

Practical Tips

- It is important that the test team is informed of all changes made to the system under test.
  Even small changes that are hidden from the user can cause an automated test execution to fail.
- The effort for a well-structured model design will now be rewarded and the ROI can be achieved (see diagram below).
  If the design is correct, only a few parts of the model need to be adapted.
  All test cases can then be automatically updated by generating them again.

When does the MBT test tool amortize (ROI)?

The initial effort is different, i.e. higher for automated tests at the beginning than for manual tests (Fig. 4). For test automation with statistical usage models, a positive balance can be achieved from the third test cycle onwards, but the achieved ROI (Return of Invest) might vary [7], [8].

In agile software development, the ROI is achieved faster through shorter iteration cycles than with classic development and quality assurance approaches.

Due to many successive development cycles, more and more functions must be tested. The increasing testing effort can only be managed by automated testing and model-based testing techniques.

Agile software development requires many iterations. Therefore, the test automation of system and acceptance tests, in addition to unit testing, is essential.

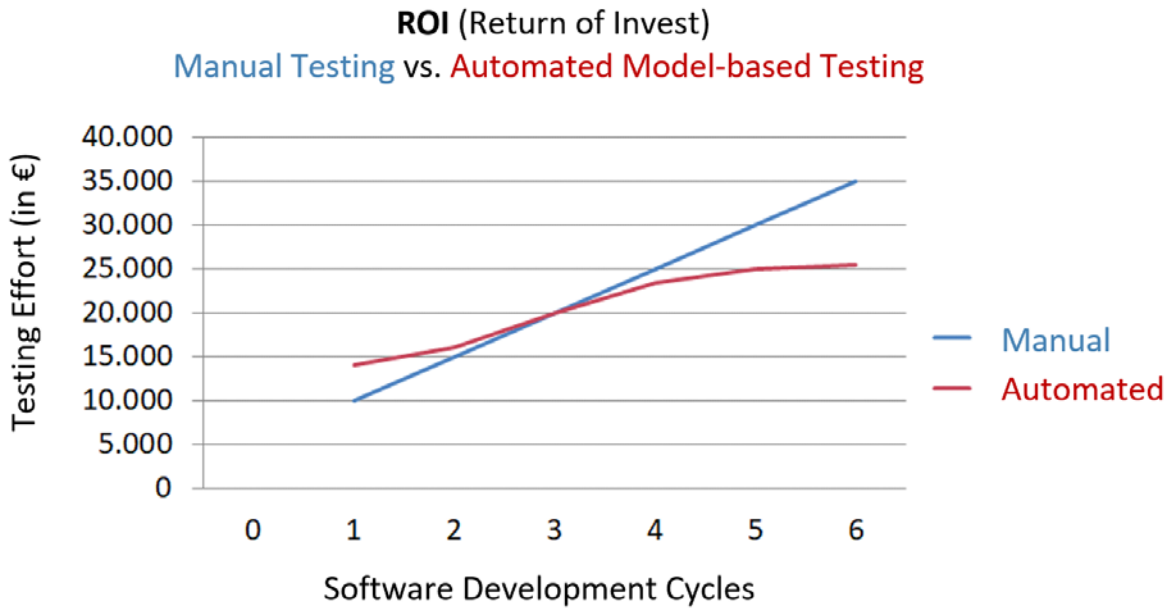


Fig. 4. ROI (Return of Invest) for automated model-based testing in agile projects

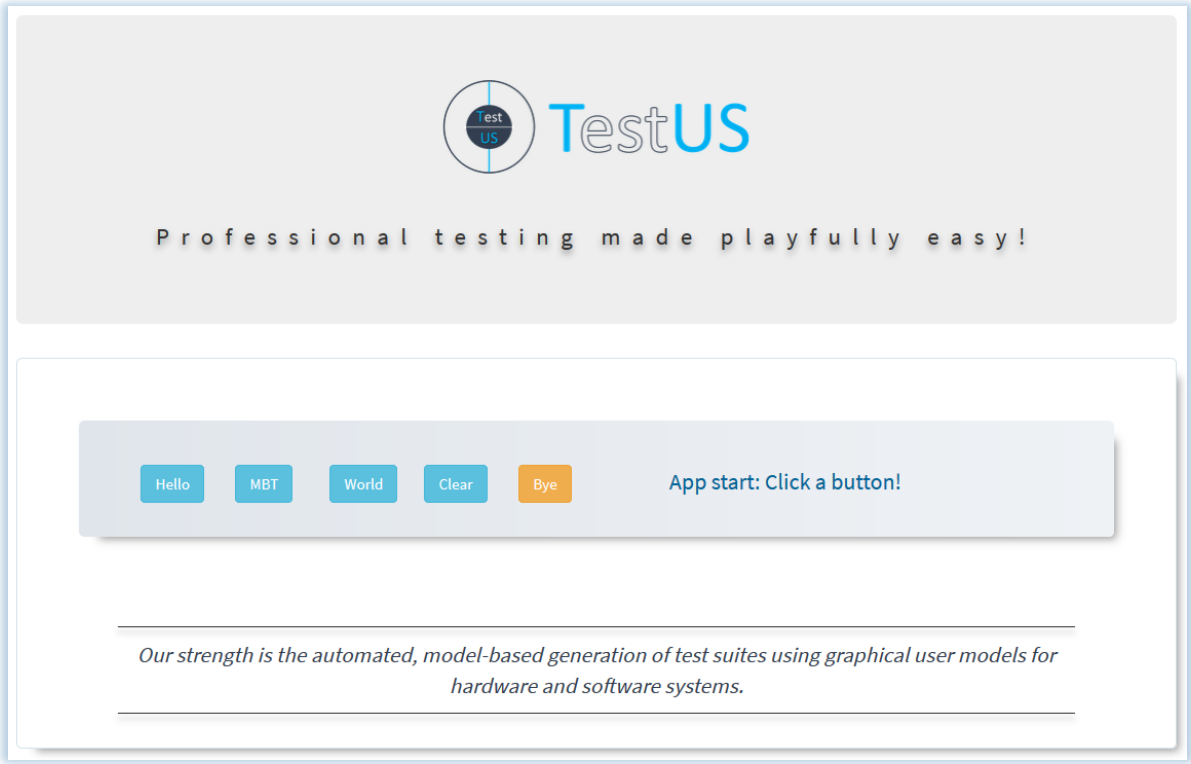
Model-based Testing of a Web Application

Our strength is the automated, model-based generation of test suites using graphical usage models for hardware and software systems. Test suites can be further processed as JSON documents for the development of executable test cases or can be visualized in a user-friendly way.

The current example HelloMBTWorld shows how model-based tests can be implemented for web applications. For this purpose, the main steps for automated statistical usage testing

- handling the test requirements
- creation of the usage model
- generation of abstract test suites
- building an executable test suite
- execution of the test
- generation of test reports

will be carried out.

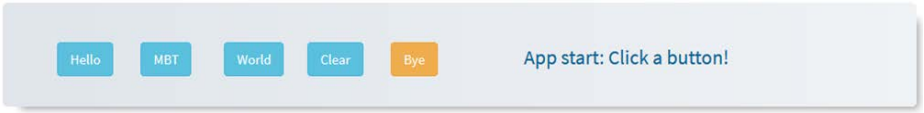


Handling the Test Requirements

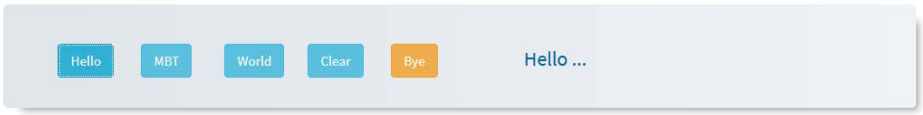
The following diagrams illustrate the required usage behavior of the web application `HelloMBTWorld`:

- when clicking on one of the three buttons `Hello`, `MBT` or `World` the corresponding text is inserted into the output field
- pressing the `Clear` button replaces the existing output text by …
- if the orange `Bye` button is clicked, the message `End of test case!` appears in the output field

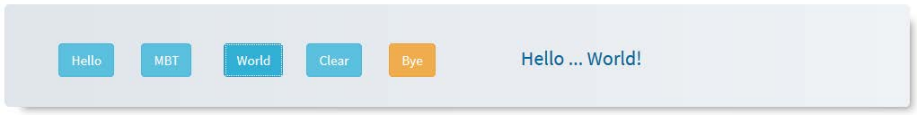
- `Start`



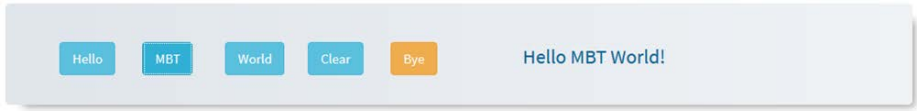
- `Hello`



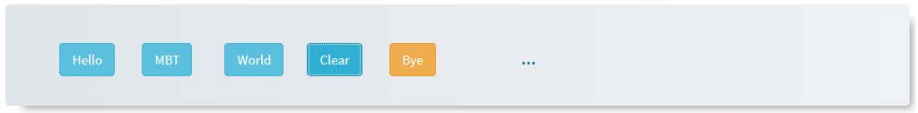
- `World`



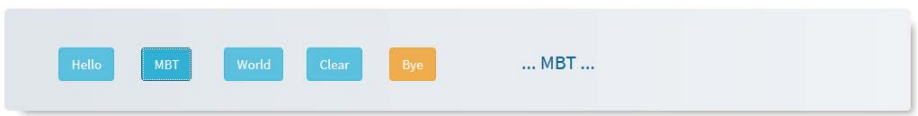
- `MBT`



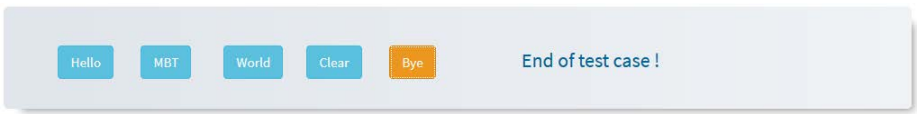
- `Clear`



- `MBT`

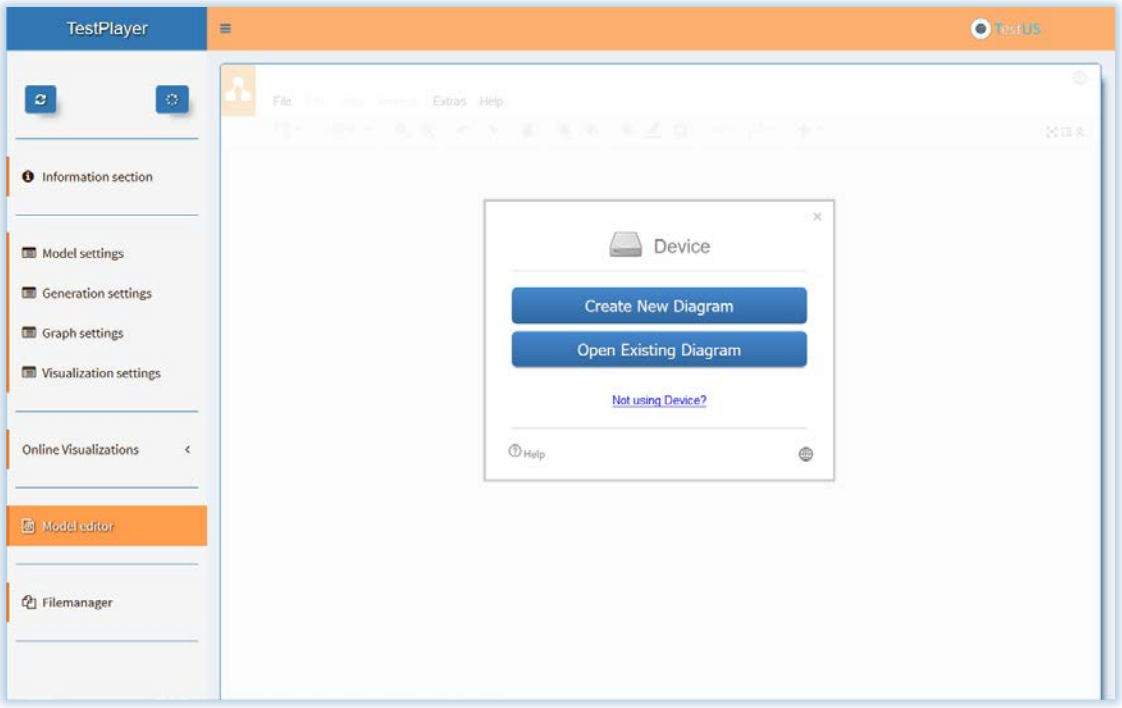


- `Bye`



Creation of the Usage Model

Usage models can be created in the `Model editor` section of the TestPlayer© Dashboard [1] by means of a graphical editor.

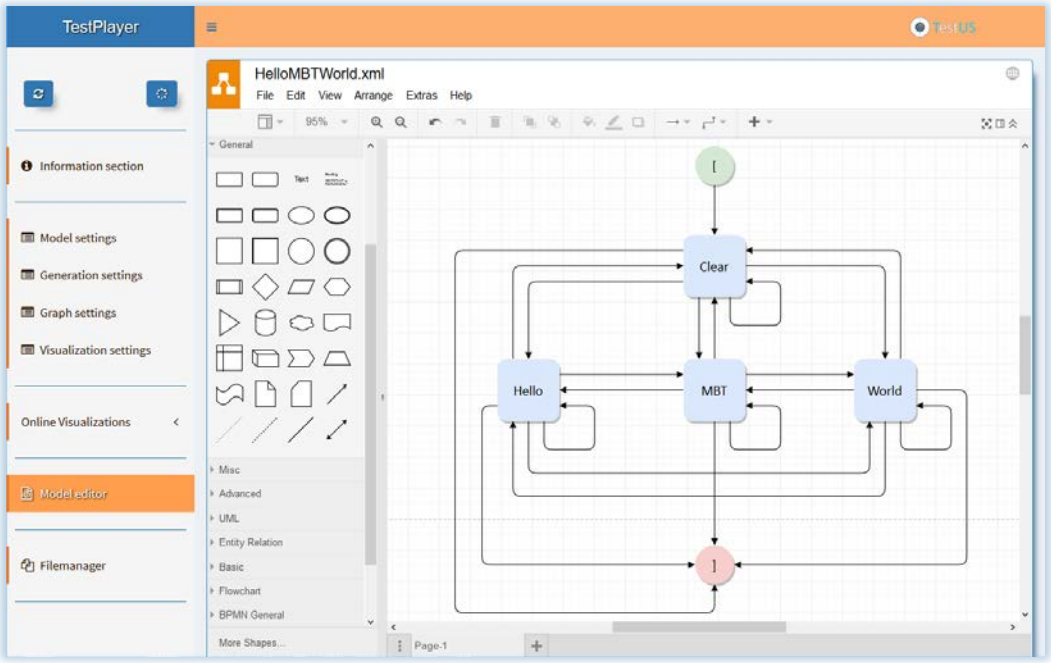


First, we model the behavior and actions of users when they interact with the web application `HelloMBTWorld`. Therefore, the statistical usage model contains a start state `[`, the state `Clear` (Clear button), the state `Hello` (Hello button), the state MBT (MBT button), the state `World` (World button), and the end state `]` that is reached when the user is pressing the Bye button.

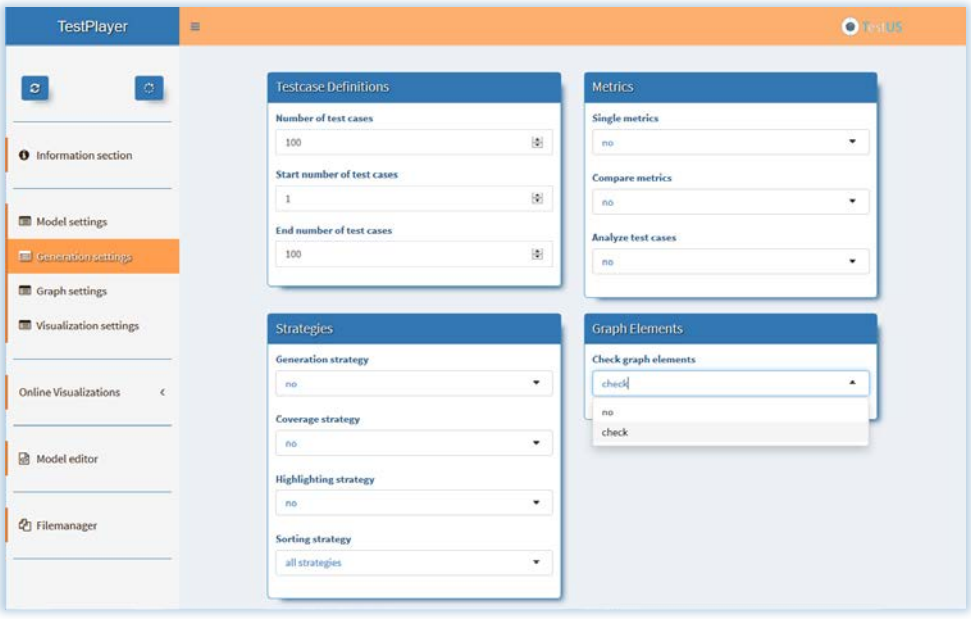
In addition, edges between the usage states must be added to describe the dynamic usage behavior. The first result we get is an in-complete usage model that still lacks the edge labels. Edge labels are pairs `(event, probability)` containing the names of the transition events and the corresponding transition probability.

In our example, the transition events are called `StartApp` (calling the web application `HelloMBTWorld`), `ClickClear` (pressing the Clear button), `ClickHello` (pressing the Hello button), `ClickMBT` (pressing the MBT button), `ClickWorld` (pressing the World button) and `ClickBye` (pressing the Bye button).

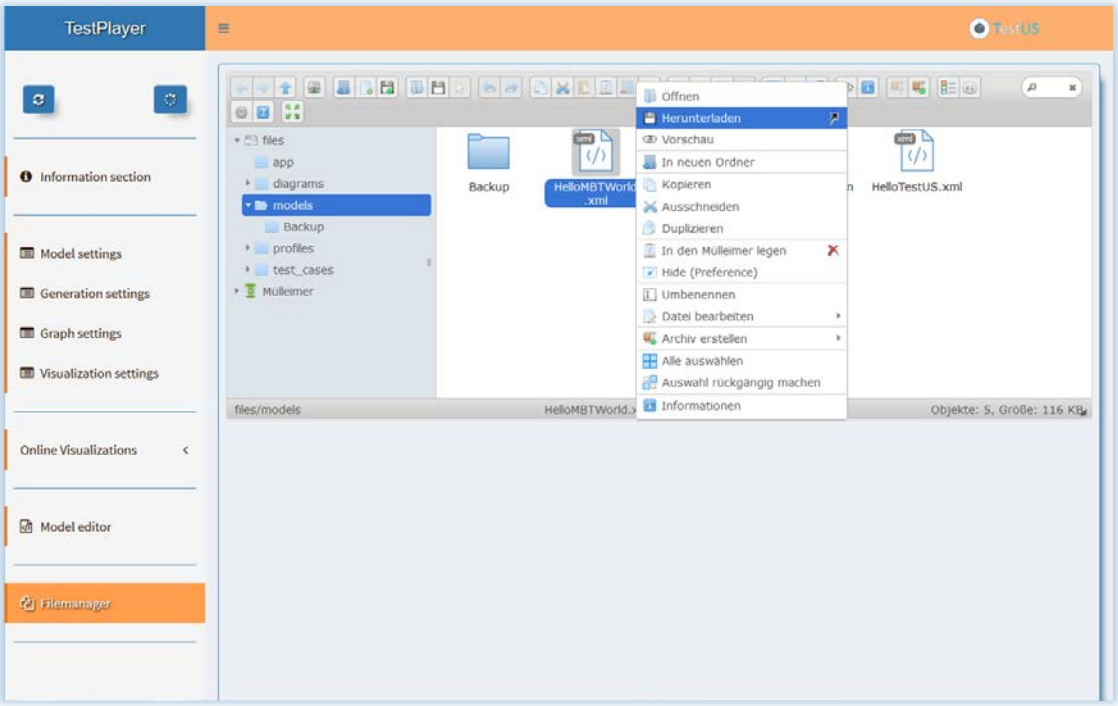
This will result in the following incomplete usage model:



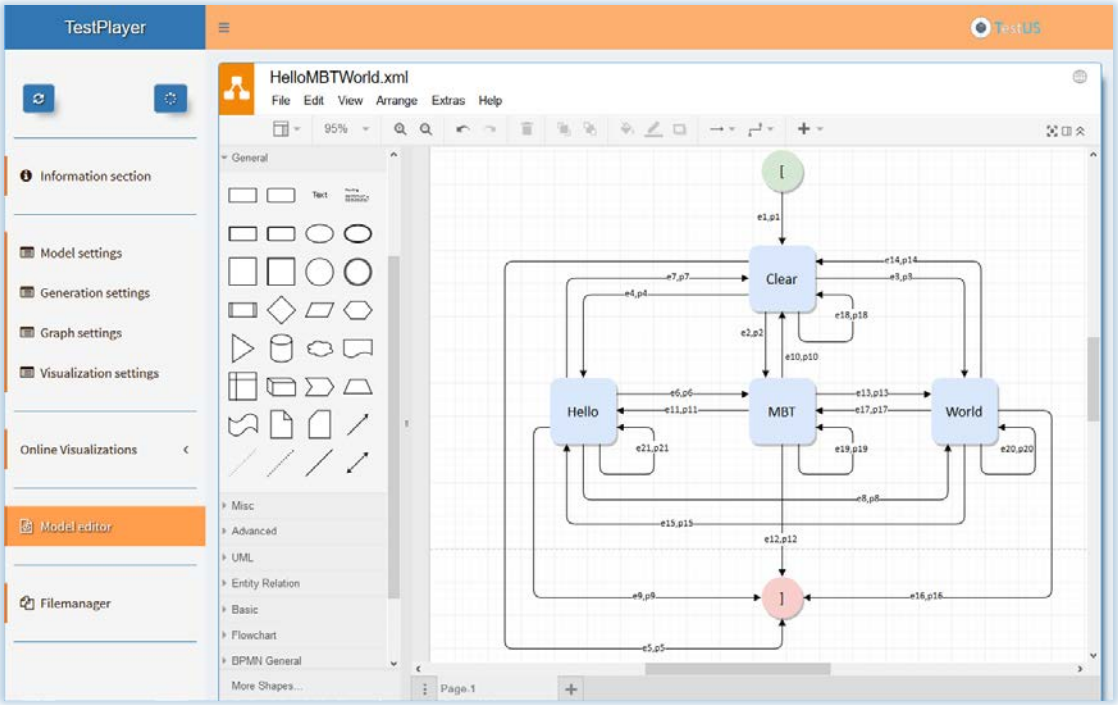
To be able to automatically generate test cases from the previous incomplete usage model without edge labels, all transition events that lead to a change of the usage states and the associated transition probabilities must be added to the edges of the usage model. The TestPlayer© automatically takes over the necessary completion by selecting the `check` option for `Check graph elements` in the dashboard:



Pressing the `Execute` button  of the TestPlayer© Dashboard will execute the selected options and create a complete usage model that can be downloaded within the Dashboard `Filemanager` section:



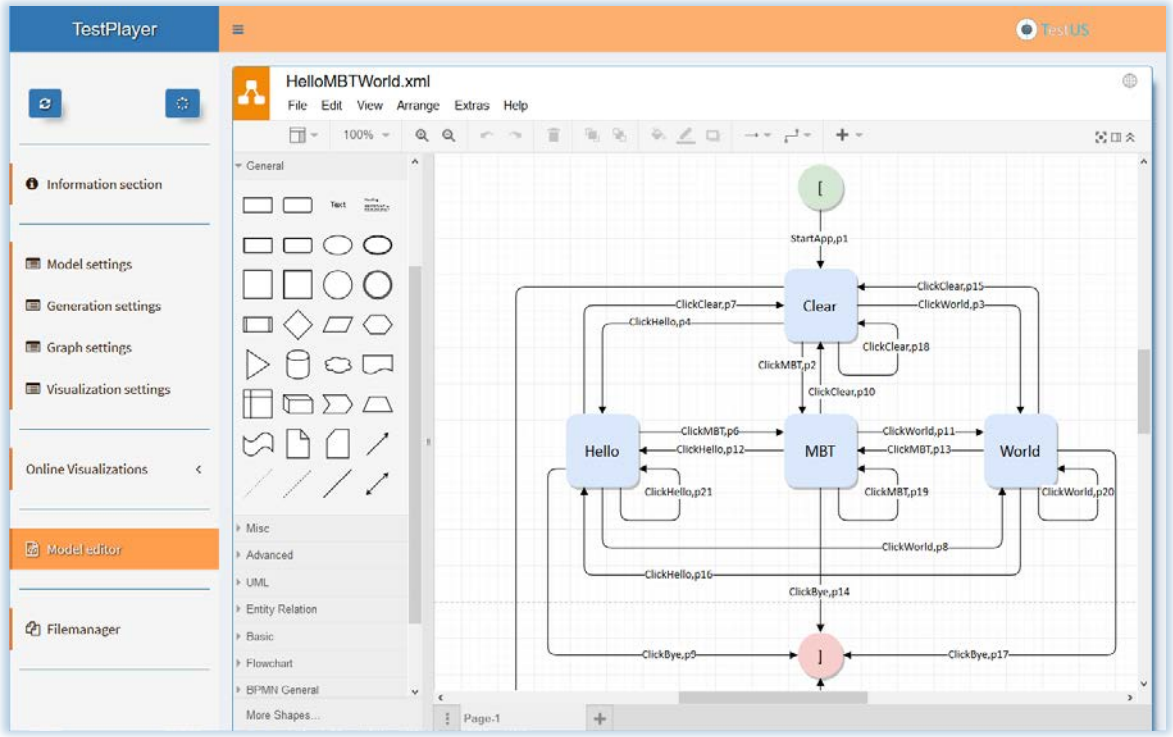
The complete usage model now contains edge labels, i.e. generic transition events that lead to a change of the usage states and the associated transition probabilities:



In the last step of the modeling process, the generic event names are adapted to the specific application names, i.e.

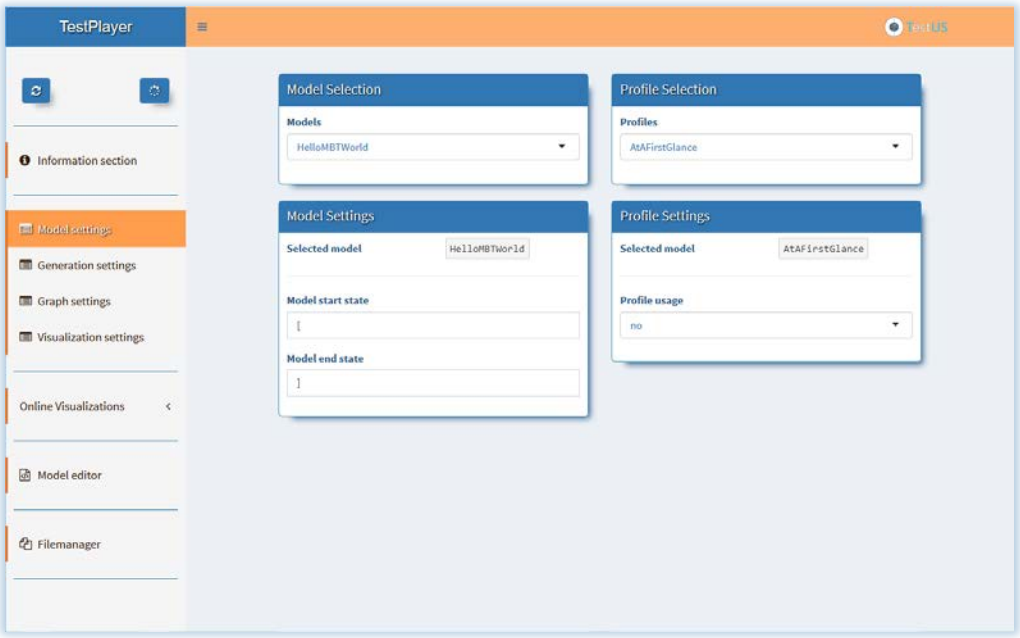
- `e1 → StartApp`
- `e2 → ClickMBT`
- `e3 → ClickWorld`
- `e4 → ClickHello`
- etc.

Due to this adaptation, the final complete usage model that contains all necessary information to automatically generate test cases for of the web application `HelloMBTWorld` looks like this:

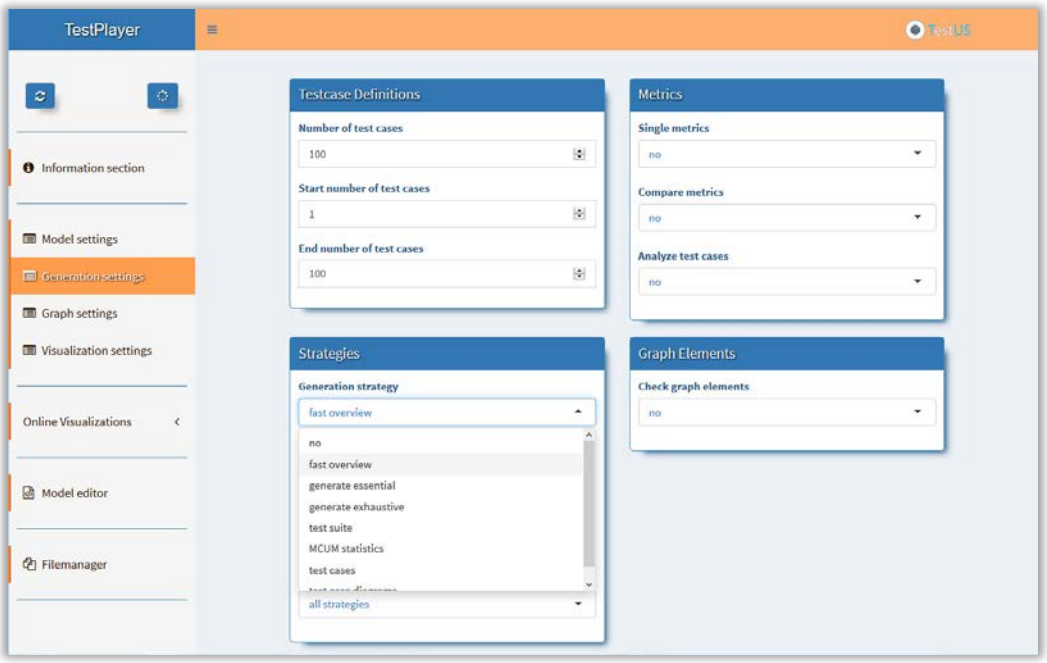


Generation of Abstract Test Suites

After uploading the usage model to the `models` section of the TestPlayer© dashboard, the automatic generation of test suites can be performed. First, `HelloMBTWorld` and the start and end states (default: `[`, respectively `]`) must be selected:

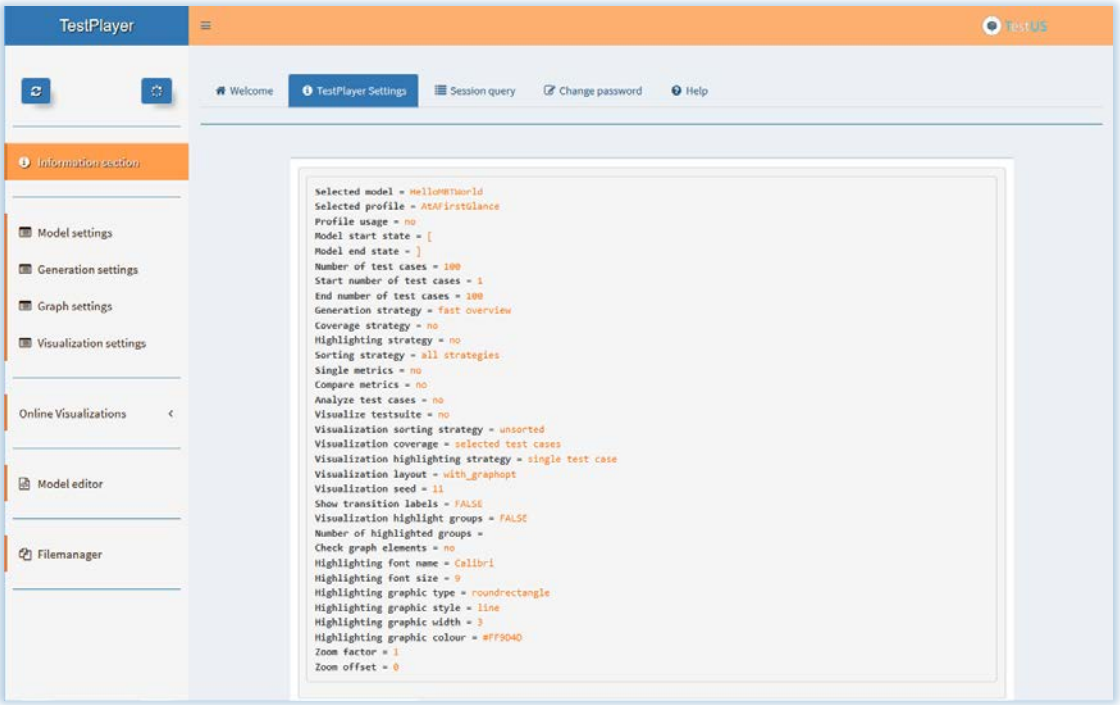


After that, the `Number of test cases` for the generated test suites (default: 100) and the `Generation strategy` (default: `fast overview`) are chosen:

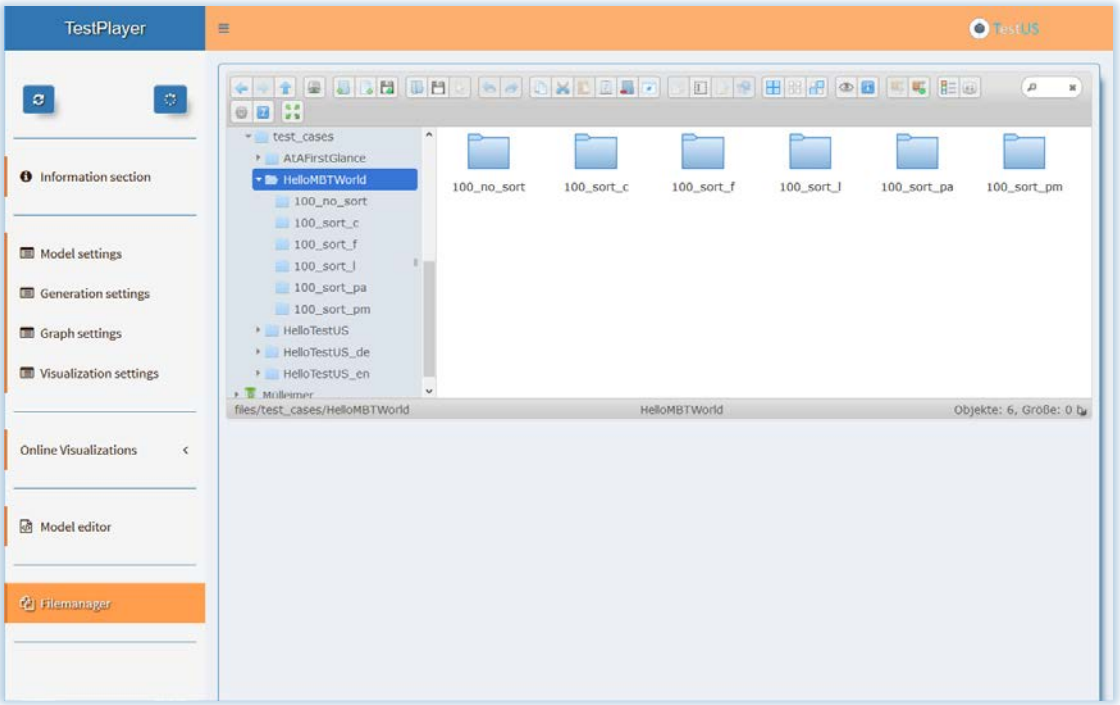


A closer look at the `TestPlayer© Settings` in the `Information` section shows the chosen generation alternatives:

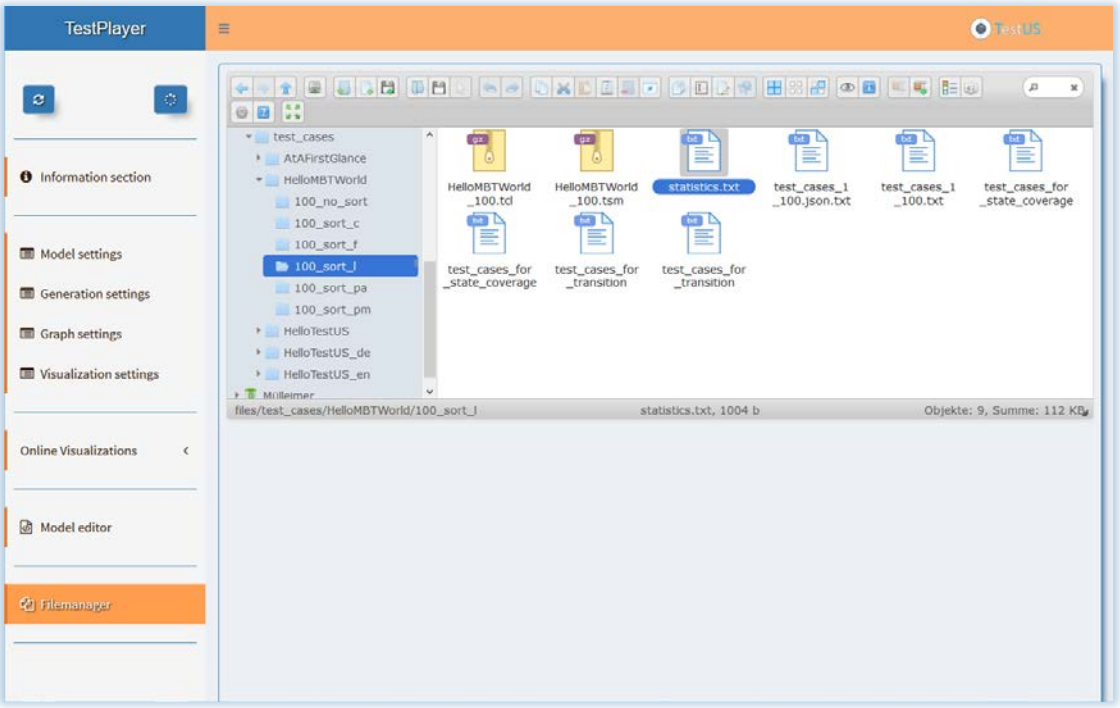
- `Selected model = HelloMBTWorld`: the test suite is generated from the usage model `HelloMBTWorld`
- `Profile usage = no`: test cases are randomly generated according to a geometric distribution without using a special usage profile
- `Number of test cases = 100`: the test suite contains 100 different test cases after generation
- `Generation strategy = fast overview`: in addition to the generation of a test suite for the specified sort criterion in the `test_cases` directory of the file manager, additional diagrams and test case visualizations are created in the `diagrams` directory of the file manager
- `Sorting strategy = all strategies`: test suites are generated for all sorting criteria [2], i.e. `no_sort` (random sort), `sort_c` (sorted by complexity), `sort_f` (sorted by frequency), `sort_l` (sorted by length), `sort_pa` (sorted by additive probabilities), `sort_pm` (sorted by multiplicative probabilities)



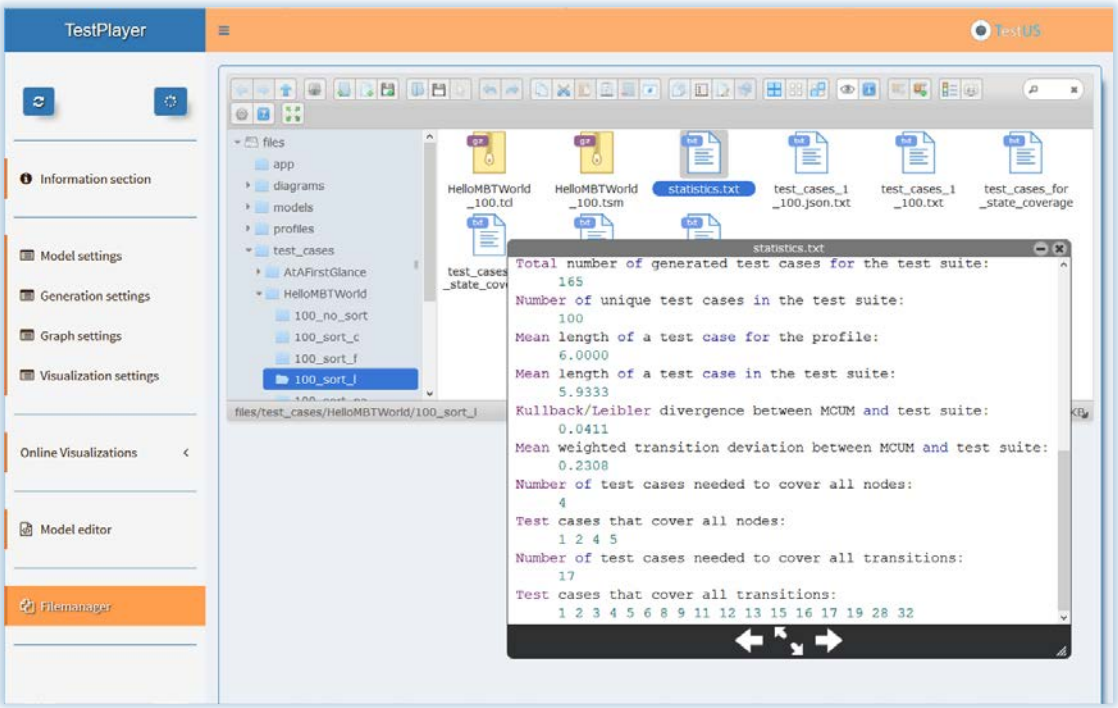
Pressing the `Execute` button  will execute the selected options and create the test cases in the directory `test_cases` as shown in the next diagram:



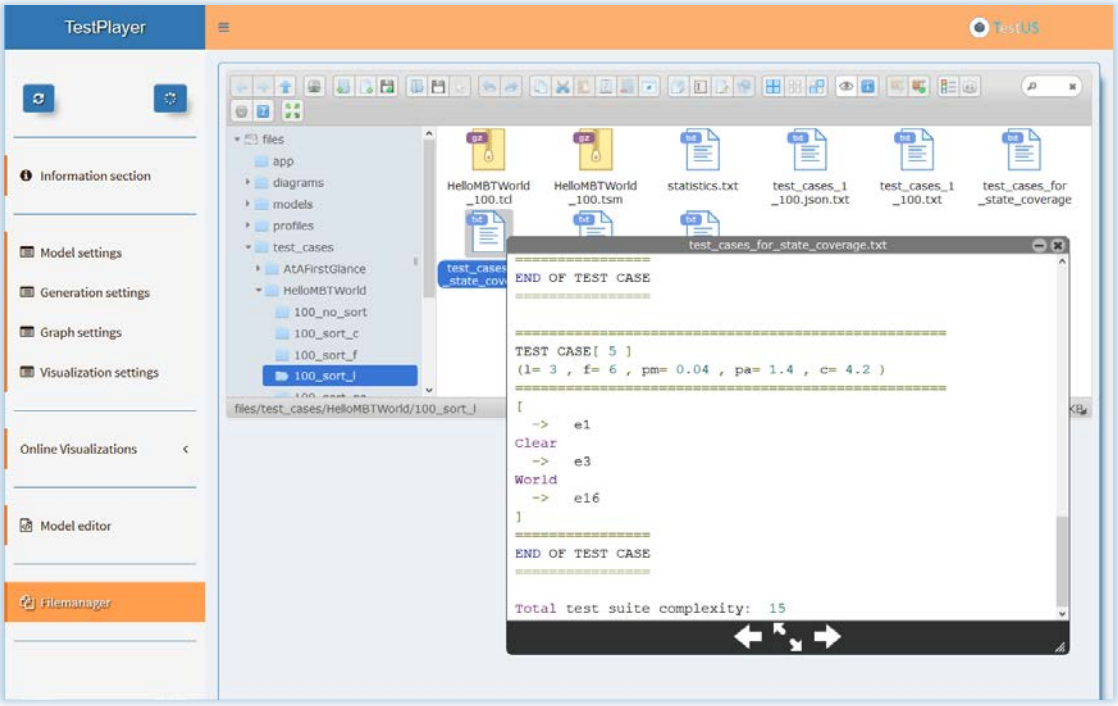
The following diagrams are results for the sorting strategy length, i.e. test cases are sorted by length (shorter first):



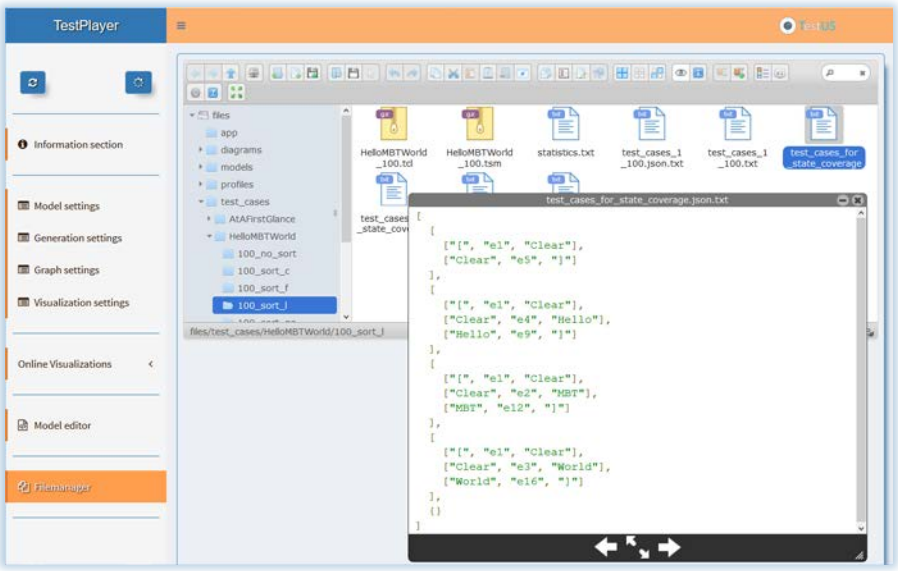
File `statistics.txt` contains additional information about the test suite, such as the number of states (nodes) and state transitions of the usage model, the mean length of a test case in the test suite, or the number of test cases that cover all states of the usage model:



File `test_cases_for_state_coverage.txt` contains test cases that are needed to cover all states of the usage model, as well as some additional information:

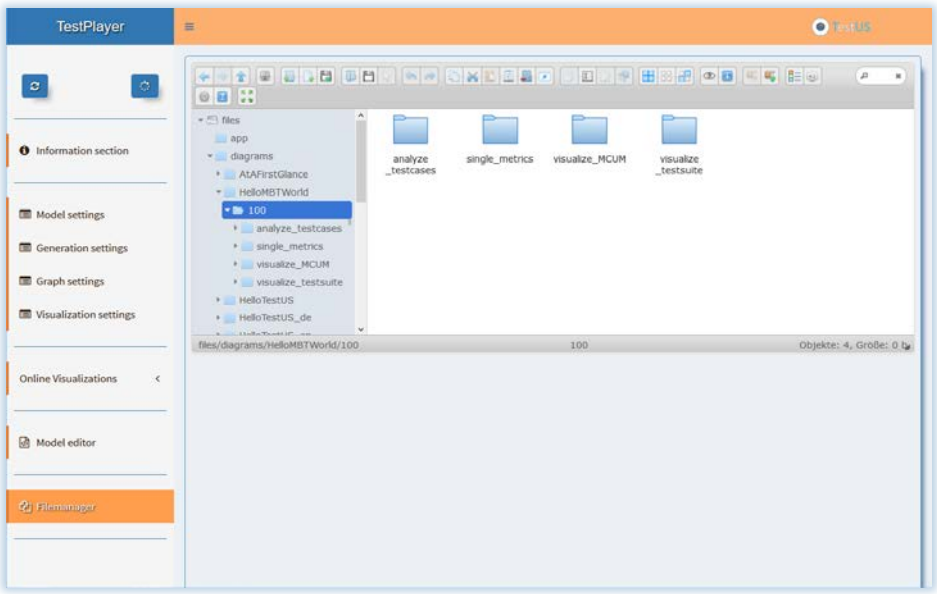


File `test_cases_for_state_coverage.json.txt` contains all the test cases that cover the states of the usage model, represented in a compact JSON format (*JavaScript Object Notation*) for further processing, e.g. for executing during the test:



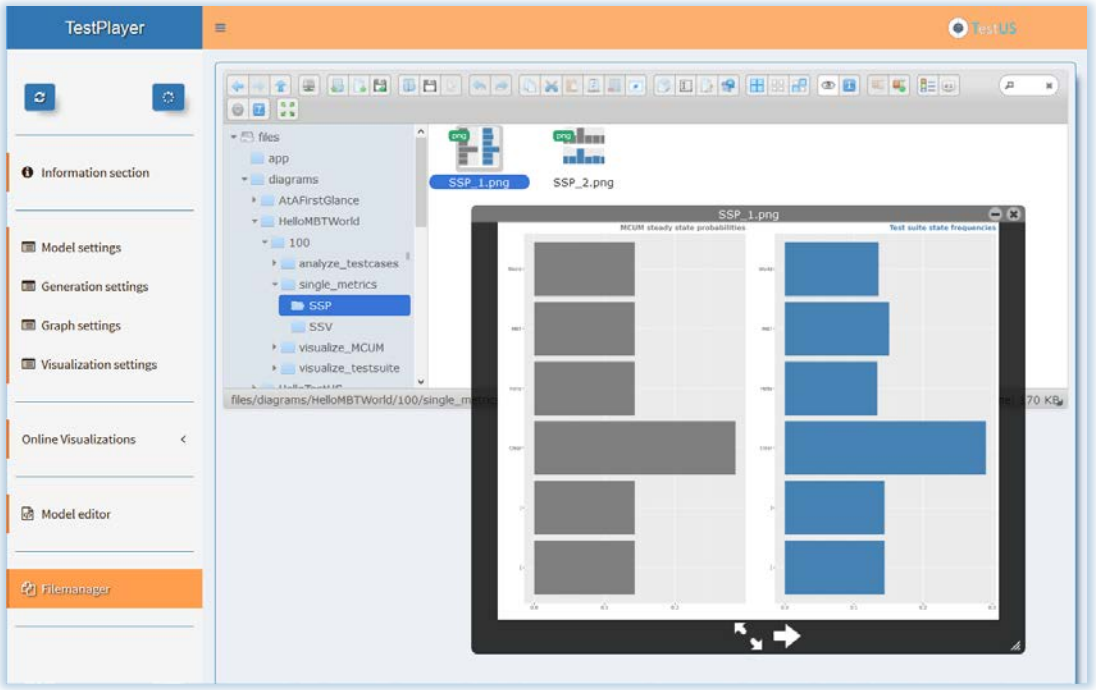
Directory diagrams contains further subdirectories for displaying the characteristic properties of the test suite:

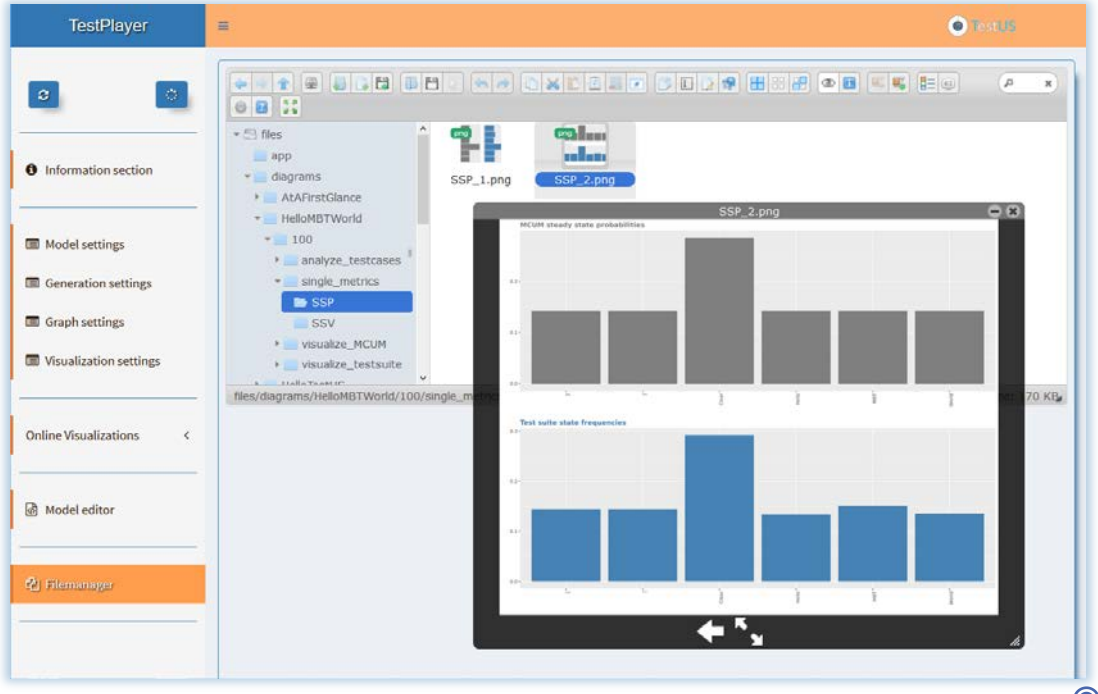
- `analyze_testcases`: various statistics for transition events and usage states
- `single_metrics`: steady state distributions of usage states and visit frequencies
- `visualize_MCUM`: XML files that contain the generated test cases to visualize the different coverage criteria of the usage model
- `visualize_testsuite`: PNG files that visualize the test suite in various



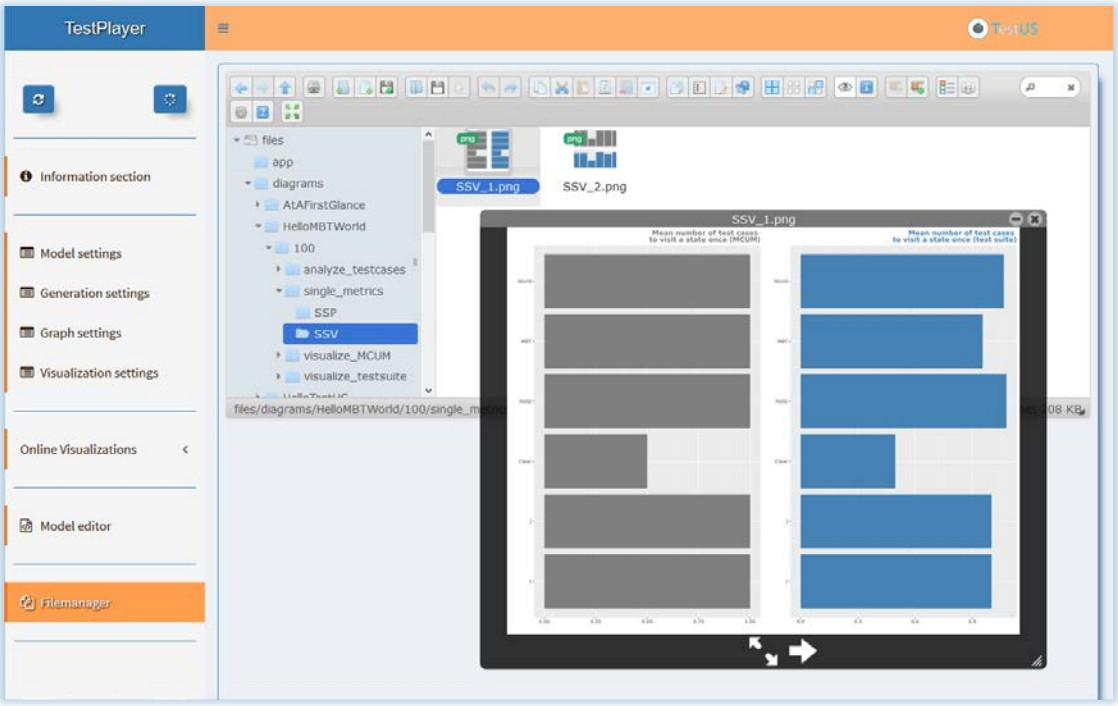
Special metrics in directory single_metrics can be used to analyze the characteristic properties of the test suite:

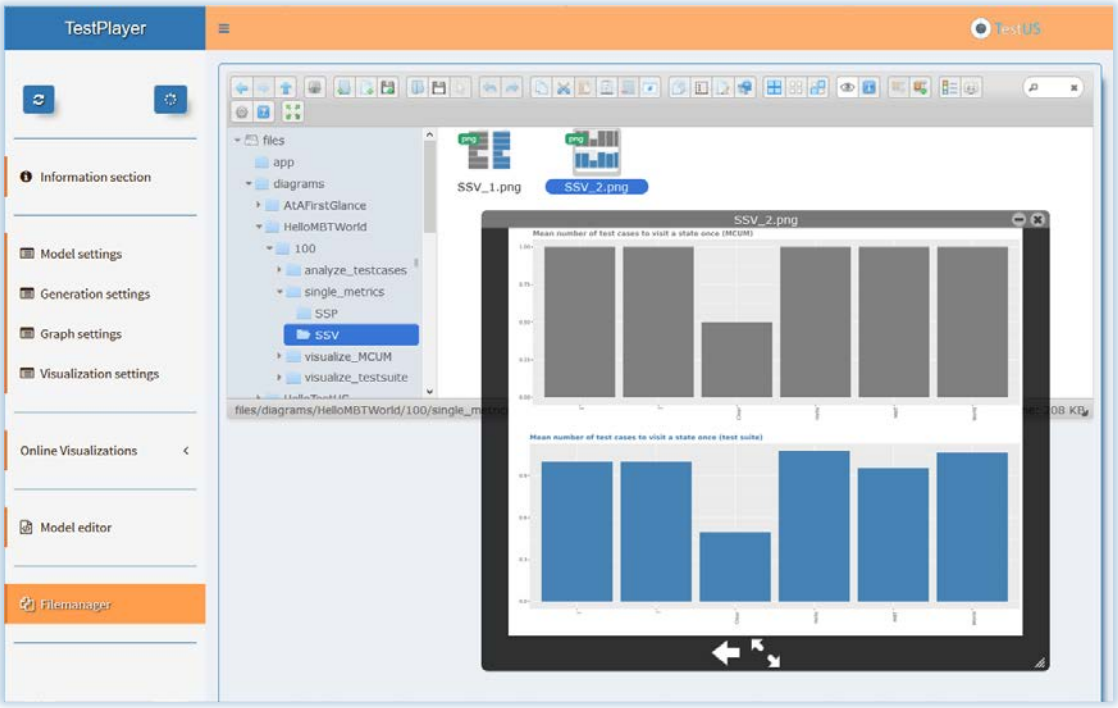
- SSP: comparison of the probability distribution of usage states in statistical equilibrium for the usage model and the relative frequencies of the corresponding usage states in the generated test suite.
  - As you can see, the values for the individual usage states are matched very well.





- SSV: comparison of the average number of test cases required to visit a state in the usage model respectively during the test execution.
  - As you can see, the values for the individual usage states are matched very well.

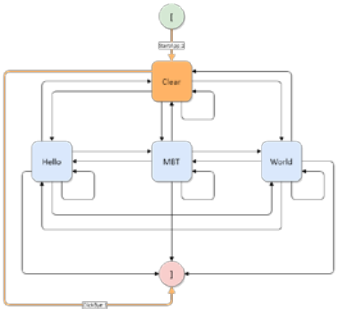




After the analysis of the test suite, the resulting test cases can be visualized. The test suite contained in file `test_cases_for_state_coverage.json.txt` consists of four test cases and covers all usage states. Test cases are sorted according to their length and shown in JSON notation too. The individual test cases are visualized (emphasized by bold orange coloring) and show the current coverage of usage states as well as transitions between the states (represented by pale orange coloring). The number after the colon of the click event shows how often the state transition within the test case was executed.

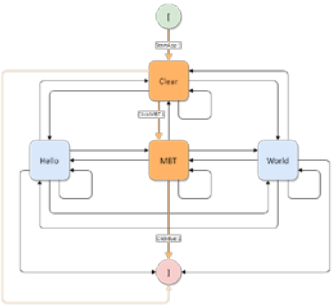
▷ Test case TC1:

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



▷ Test case TC2:

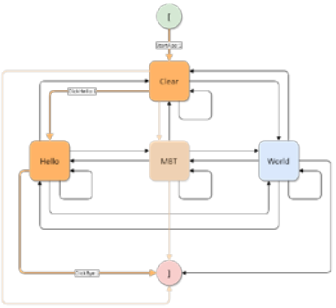
```
[
   ["[", "StartApp", "Clear"],
   ["Clear", "ClickMBT", "MBT"],
   ["MBT", "ClickBye", "]"]
```



▷ Test case TC3:

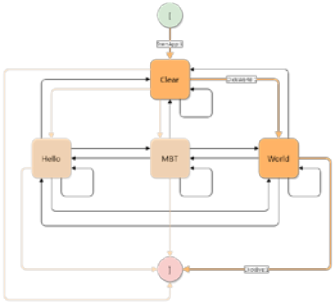
```
[
   ["[", "StartApp", "Clear"],
   ["Clear", "ClickHello", "Hello"],
   ["Hello", "ClickBye", "]"]

 ]
```



▷ Test case TC4:

```
[
   ["[", "StartApp", "Clear"],
   ["Clear", "ClickWorld", "World"],
   ["World", "ClickBye", "]"]
 ]
```



Building an Executable Test Suite

Eclipse is an open-source programming environment for modeling and developing all kinds of (application) software, which fits ideally into a comprehensive test framework in combination with the TestPlayer©. 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

Automated testing of web applications requires additional drivers to provide the ability to automatically access the respective web browser. We use the test automation framework Selenium [5], which can be easily integrated into an Eclipse- based test environment and offers a common Java API for the main web browsers (Fig. 5).

```
// ...
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();
}
// ...
```

Fig. 5: Elements of the Selenium web driver Java API for Eclipse.

For web applications to be tested automatically, a testing interface must be provided that simulates the state-based logic of the usage model. For this purpose, each test step describes a state transition that implements the desired test request. The Java `switch()` statement in Fig. 6 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, method `Thread.sleep(time)` is performed in addition.

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>
```

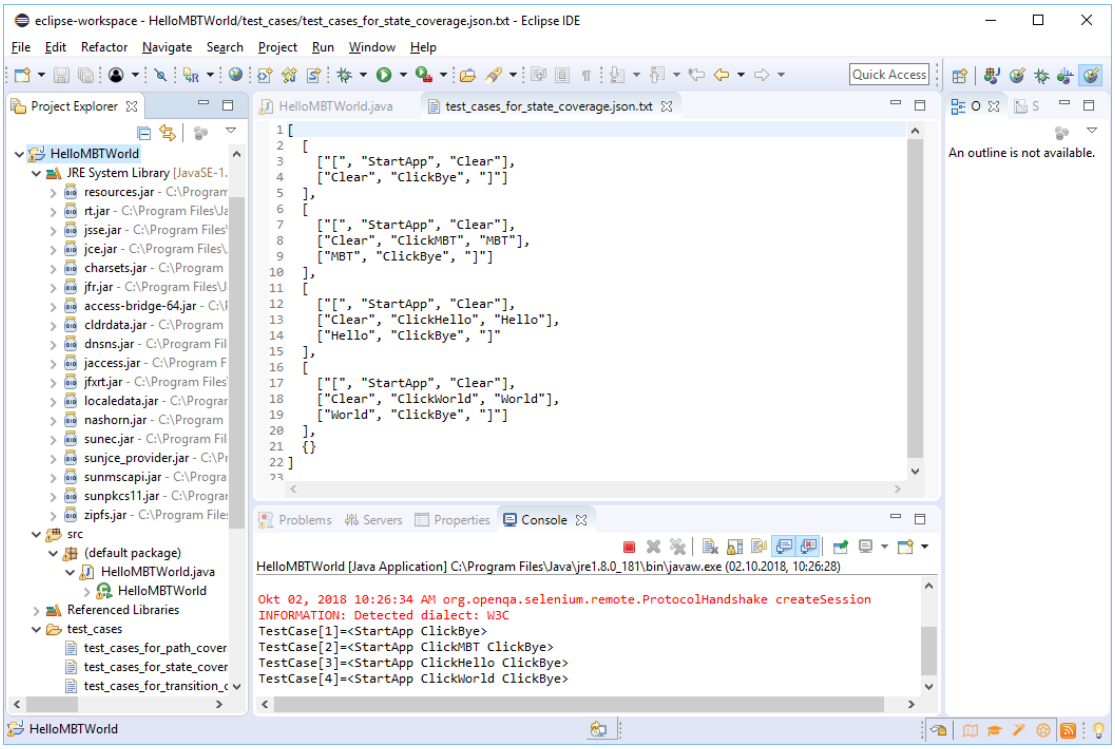
```
for (String key : keyClicks) {
  switch (key) {
    Case "ClickHello":
      byID("Hello");
      Thread.sleep(time);
    break;
    case "ClickMBT":
      byID("MBT");
      Thread.sleep(time);
    break;
    // ...
  }
}
```

Fig. 6: Java `switch()` programming pattern for executing test steps.

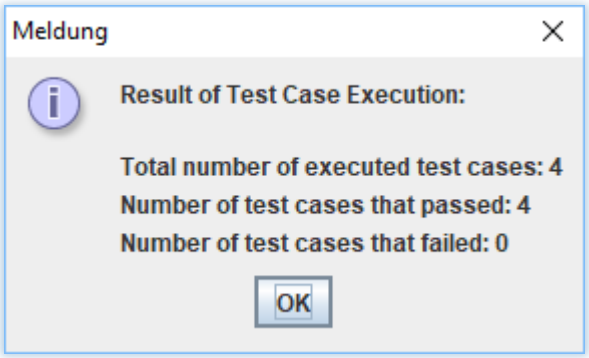
Before testing a web application, the test engineer must first select the type of the web browser to start the correct driver. Concrete values for the driver are `FirefoxDriver()` for the Mozilla web browser and `ChromeDriver()` for the Google web browser. The web application can then be started via `startApp(String URL)` method for the given URL and subsequently automatically tested using the previously generated test cases.
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 ID or `byTag(String tag)` for clicking the next HTML element with the given HTML tag (Fig. 5).

Execution of the Test

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



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



Model-based Testing of 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 of websites. For this purpose, are using the TestUS Homepage to show how any web presence can be tested with model-based techniques [3]. For the automatic generation of the test suite, we first must create a proper usage model for the website by means of the TestPlayer© Dashboard. For the TestUS homepage, the corresponding usage model is as shown in Fig. 7:

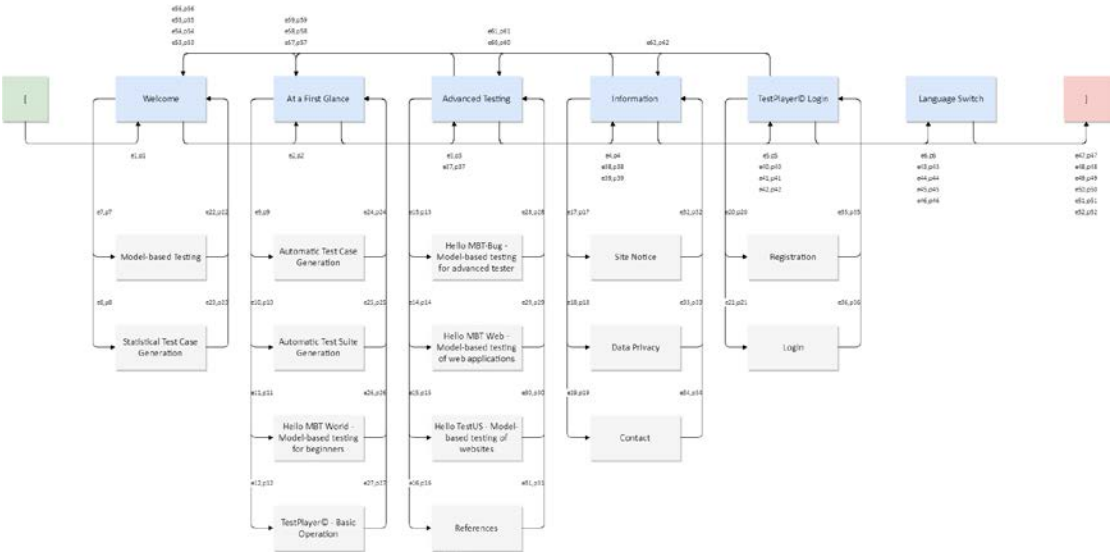


Fig. 7: Usage model of the TestUS website.

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

- Starting in `Welcome`
  - `Model-based Testing` and
  - `Statistical Test Case Generation` are reached
- `At a First Glance` provides access to
  - `Automatic Test Case Generation`,
  - `Automatic Test Suite Generation`,

- Hello MBT World - Model-based testing for beginners,
- TestPlayer© - Basic Operation

▷ Advanced testing offers access to

- Hello MBT Bug - Model-based testing for advanced tester,
- Hello MBT Web - Model-based testing of web applications,
- Hello TestUS - Model-based testing of websites and
- References

▷ Information gives you access to

- Site Notice and
- Contact

▷ TestPlayer© Login offers the access to

- Sign-Up and
- Login

▷ Language Switch  , respectively  switches the language between English, respectively German

State transitions between the usage states are triggered by the generic events e1 to e62, which were automatically generated by the TestPlayer© together with the corresponding transition probabilities p1 to p60.

Using the TestPlayer© it is now possible to automatically generate test cases for the model-based test of the TestUS homepage from the given usage model presented in Fig. 7. As a result, the test suite shown in Fig. 8 contains 21 test cases for complete coverage of all usage states, which are created according to the sorting criterion sort_l, i.e. the test cases are selected from 100 statistically random generated test cases sorted according to the length of the test cases.

The test cases are animated (marked by bold orange coloration) and show the coverage of the usage states as well as the transitions between the states. The number after the colon of the respective click event shows how often the respective state transition was executed during the test case. The visualization in Fig. 8 shows only the last three out of 21 test cases that completely cover the usage states of the TestUS website.

Fig. 8: Last three out of 21 test cases that cover the usage states of the TestUS website.

In addition to the graphical representations, the TestPlayer© also generates text files that provide the test cases of the test suite in a compact JSON format for test execution. The file `test_cases_for_state_coverage.json.txt` contains the 21 test cases of the test suite `state_coverage` for testing the TestUS homepage. Following you will find just the first two and the last three of 21 test cases that cover the usage states of the TestUS website.

```
[
 ["[", "e1", "Welcome"],
 ["Welcome", "e52", "]"]
]

[
 ["[", "e1", "Welcome"],
 ["Welcome", "e3", "Advanced Testing"],
 ["Advanced Testing", "e50", "]"]
]
          ...
[
 ["[", "e1", "Welcome"],
 ["Welcome", "e2", "At a First Glance"],
 ["At a First Glance", "e9", "Automatic Test Case Generation"],
 ["Automatic Test Case Generation", "e24", "At a First Glance"],
 ["At a First Glance", "e51", "]"]
]

[
 ["[", "e1", "Welcome"],
 ["Welcome", "e3", "Advanced Testing"],
 ["Advanced Testing", "e14", "Hello MBT Web - Model-based testing of web applications"],
 ["Hello MBT Web - Model-based testing of web applications", "e29", "Advanced Testing"],
 ["Advanced Testing", "e44", "Language Switch"],
 ["Language Switch", "e47", "]"]
]

[
 ["[", "e1", "Welcome"],
 ["Welcome", "e3", "Advanced Testing"],
 ["Advanced Testing", "e13", "Hello MBT-Bug - Model-based testing for advanced tester"],
 ["Hello MBT-Bug - Model-based testing for advanced tester", "e28", "Advanced Testing"],
 ["Advanced Testing", "e44", "Language Switch"],
 ["Language Switch", "e47", "]"]
]
```

Automated testing of the TestUS website requires additional drivers to provide the ability to access the respective web browser. Again, as presented in the previous `HelloMBTWorld` example, we use the test automation framework Selenium, which offers a common Java API for the main web browsers (Fig. 5). We must also adapt the interface to refer to IDs that are used as input parameters for method `byID ()` and which are the corresponding HTML id attributes in the index.html file of the website as shown in Fig. 9. For example, `ID N7_en` is used to access the usage state `Model-based Testing` during the test execution.

```
<div class="rd-navbar-nav-wrap">
    <ul class="rd-navbar-nav">
        <li id="N1_en"><a href="index_en.html"><span
                class="fas fa-home"></span> Welcome</a>
            <ul class="rd-navbar-dropdown">
                <li id="N7_en"><a href="N7_en.html"><span
                        class="fas fa-caret-right"></span> Model-based Testing</a></li>
                <li id="N8_en"><a href="N8_en.html"><span
                        class="fas fa-caret-right"></span> Statistical Test Case
                        Generation</a></li>
            </ul>
        </li>
                    ...
        <li id="N5_en ><a nref="index_en.html"><span
                class="fas fa-user"></span> TestPlayer&copy; Login</a>
            <ul class="rd-navbar-dropdown">
                <li id="N20_en"><a href="N20_en.html"><span
                        class="fas fa-address-card"></span> Sign up</a></li>
                <li id="N21_en"><a href="N21_en.html"><span
                        class="fas fa-user"></span> Login</a></li>
            </ul>
        </li>

        <li><a href="index_de.html"> <img id="N6_en" src="img/de.gif"
                alt="Switch to German">
        </a>
        </li>
    </ul>
</div>
```

Fig. 9: HTML IDs for accessing the corresponding usage state of the TestUS website.

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 (Fig. 10). 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 behavior 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()` (Fig.11).

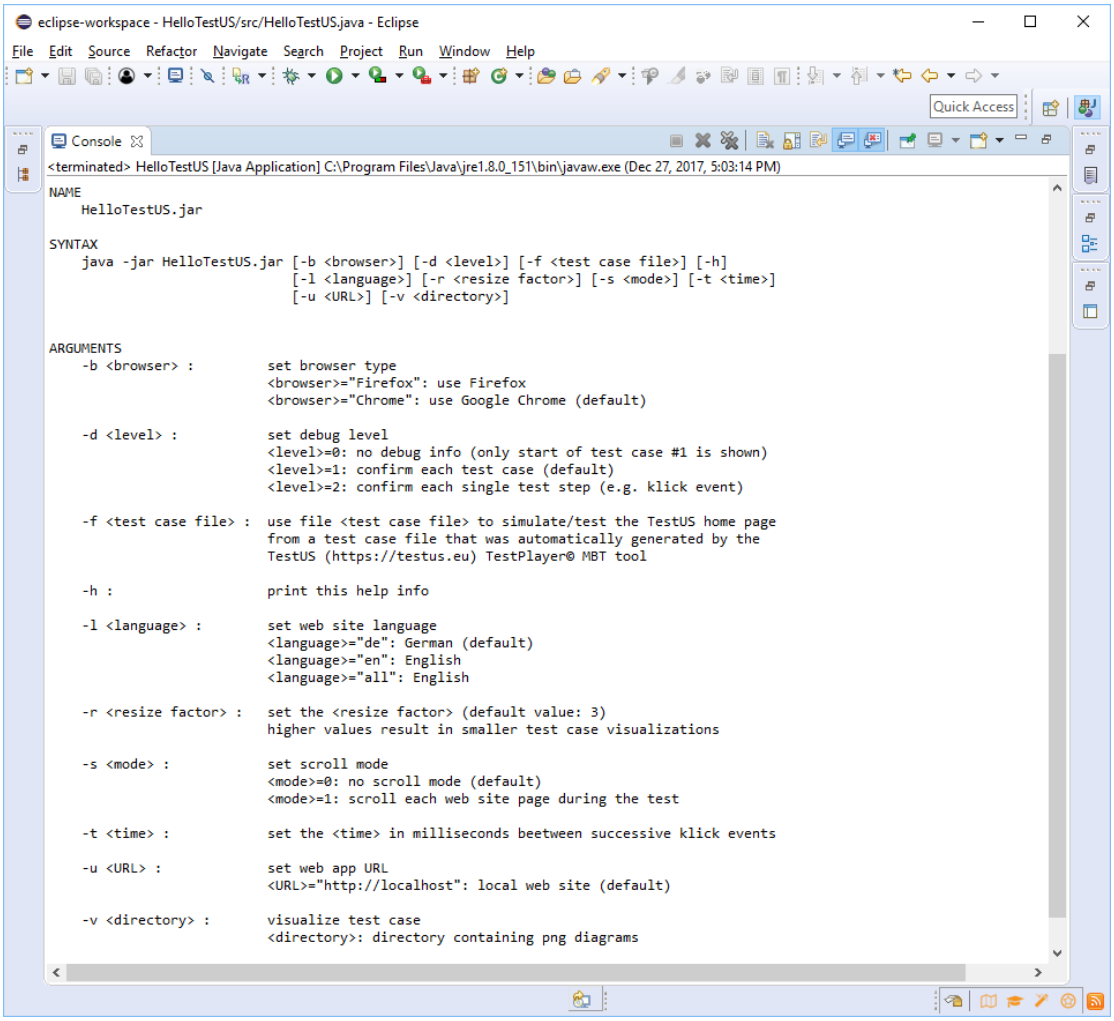


Fig. 10: Call options for testing the TestUS website.

The Java code fragment in Fig. 11 shows the actions that are triggered when selecting the usage state `Model-based Testing` within a test case, which is indicated by the generic transition event `e7`. Method `clickLinkByPartialText(N7_Text)` is used to wrap the Selenium API code `driver.findElement(By.partialLinkText(N7_Text)`. In this way it is possible to access a link that contains the string `N7_Text` while testing the website. String `N7_Text` has previously been initialized to value `Model-based Testing` when option "`-l en`" was used to test the English version of the TestUS website (Fig. 10).

The advantage of this approach is the possibility to test multilingual websites. Using of the option "`-l de`" instead enables testing of the German version of the TestUS website when String `N7_Text` has previously been initialized to value `Model-basiertes Testen`.

The advantage of this approach is the possibility to test multilingual websites. Using of the option "-l de" instead enables testing of the German version of the TestUS website when String N7_Text has previously been initialized to value Model-basiertes Testen.

```
case "e7":
    clickLinkByPartialText(N7_Text);
    // scroll part
    if (scrollMode == 1) {
        Thread.sleep(mainTime);
        for (int second = 0;; second++) {
            if (second >= 50) {
                break;
            }
            jsExecutor.executeScript("window.scrollBy(0,50)");
            Thread.sleep(100);
        }
        jsExecutor.executeScript("window.scrollTo(0,0)");
    } else
        Thread.sleep(mainTime);
    break;
```

Fig. 11: Code fragment that represents the test step for transition event `e7`.

When the generic transition event `e7` appears in a test case the TestPlayer© `Model-based Testing` item is clicked automatically in the selected web driver by performing the method `clickLinkByPartialText(N7_Text)`. After a predefined timeout of `mainTime`, which controls the web page display time, the JavaScript engine of the web browser must execute 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)`.

In Eclipse, `Run Configuration` can be used to pass the call parameters for executing the test (Fig. 12). The file option "`-f test_cases_for_state_coverage.json.txt`" imports the automatically generated test suite that covers all the usage states during the test execution. The time parameter "`-t 1000`" sets the duration of a click event during the test execution in milliseconds, i.e. one second.

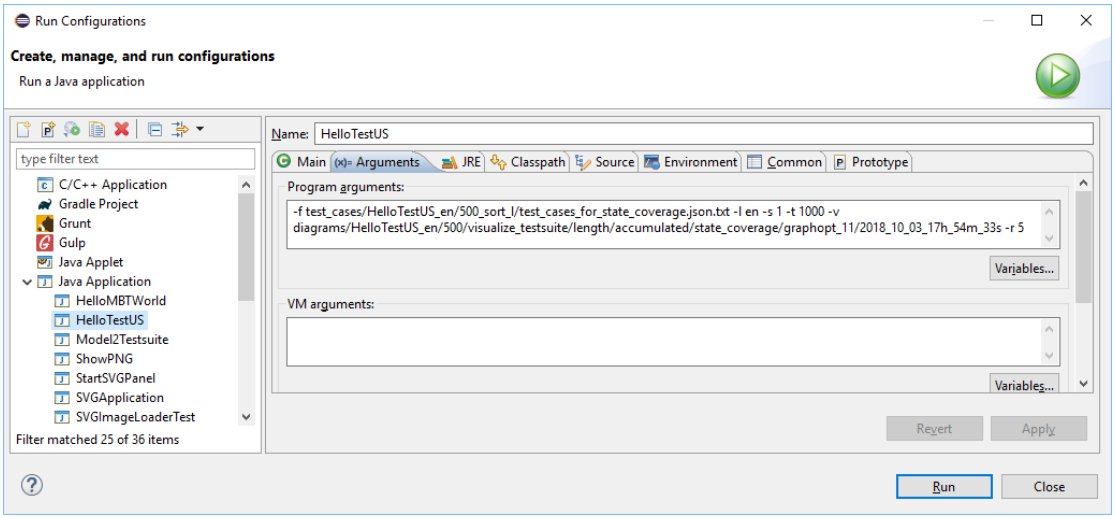


Fig. 12: Run `Configuration` for passing call parameters in Eclipse.

Subsequently, the web browser starts and automatically executes the generated click events of the imported test suite, controlled by the Chrome web driver of the Selenium framework. Call option "`-v directory`" (Fig. 12) is used to dynamically visualize each executed test case (Fig. 13). The required visualization directory can be generated automatically for this purpose within the TestPlayer© Dashboard.

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

During the test execution each test case is logged in the console output and dynamically visualized according to the preferred visualization layout. Fig. 13 shows an accumulated test case for a test suite that covers all usage states.

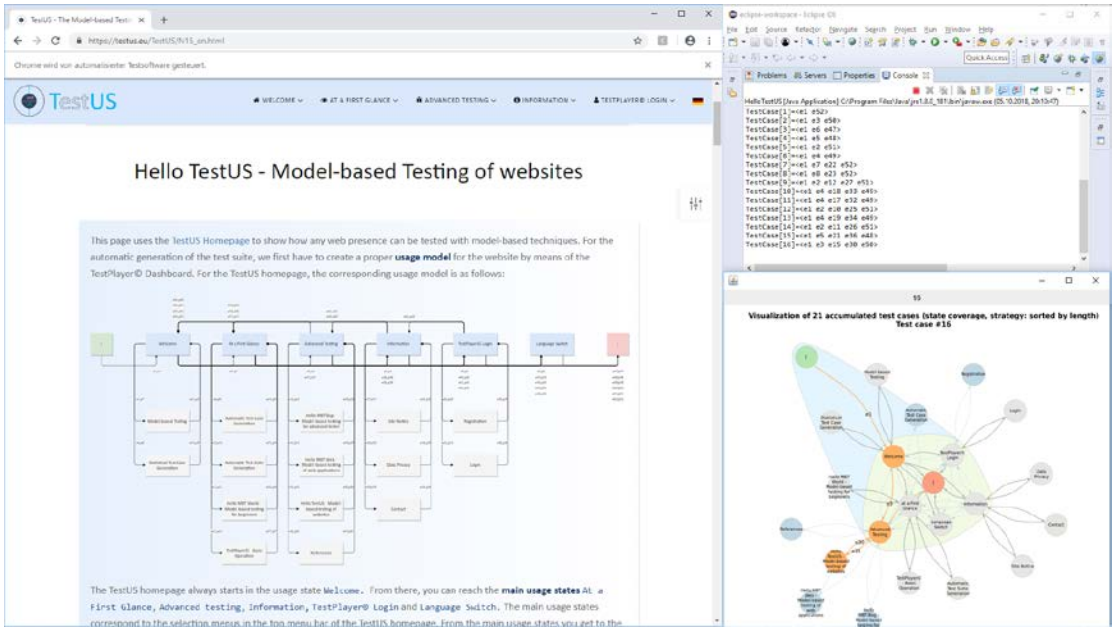


Fig. 13: Model-based test execution and test case visualization for the TestUS website.

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 (S_i,S_j) that is starting in usage state Si and ending in usage state S j by setting the corresponding probability value p(S_i,S_j) = 0
- forcing a transition (S_i,S_j) by setting the corresponding probability value p(S_i,S_j) = 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.

Figure 14 shows an adapted test suite that focuses only on those visitors of the TestUS website (Fig. 7) who access the top menu `At a First Glance`. The corresponding usage profile is as follows:

`p3=0, p4=0, p5=0, p6=0, p7=0, p8=0, p37=0, p38=0, p40=0, p43=0, p53=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.

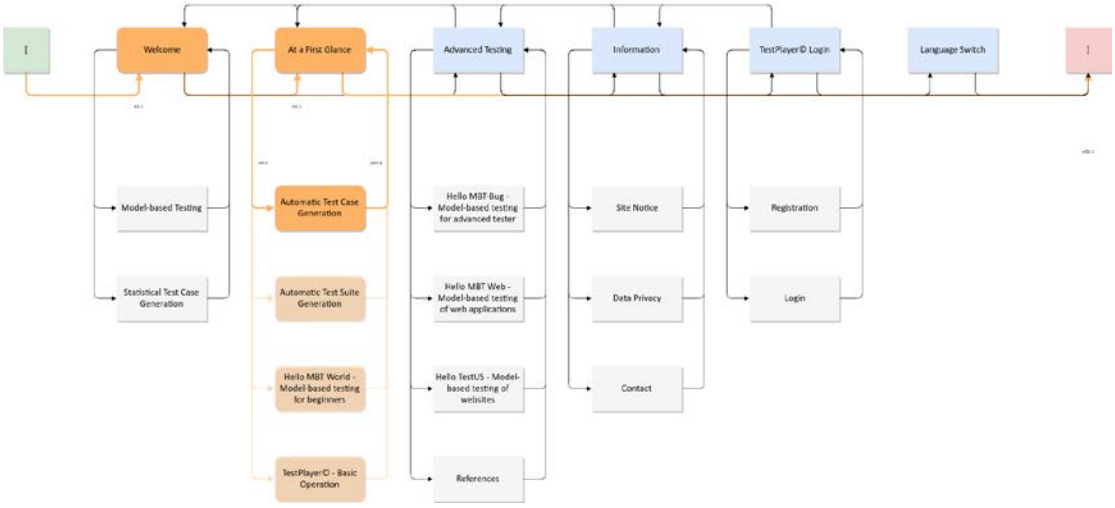


Fig. 14: Adapted test suite for the TestUS website related to a specific user profile.

Graphical Visualization of Test Cases and Test Suites

The Visualization layout determines which graphic layout for representing individual test cases of a given test suite shall be chosen. Below different layouts are shown for the same test case (last test case out of 6 that cover the transitions of the adopted test suite in Fig. 14) to compare the differences of the visualization layouts. The diagrams show an accumulated test case, i.e. already covered (visited) states and transitions are highlighted in grey and the displayed test case is shown in orange. Some interesting layout strategies are:

- simulated annealing: places vertices of the graph on the plane, according to simulated annealing (Fig. 15)
- star shaping: places one vertex in the center of a circle and the rest of the vertices equi-distantly on the perimeter (Fig. 16)
- circle shaping: places vertices on a circle, in the order of their vertex ids (Fig. 17)
- tree shaping: a tree-like layout, which is perfect for trees and acceptable for graphs with not too many cycles (Fig. 18)
- force-directed layout: layout algorithm that scales relatively well to large graphs (Fig. 19)
- rectangular grid: places vertices on a two-dimensional rectangular grid (Fig. 20)

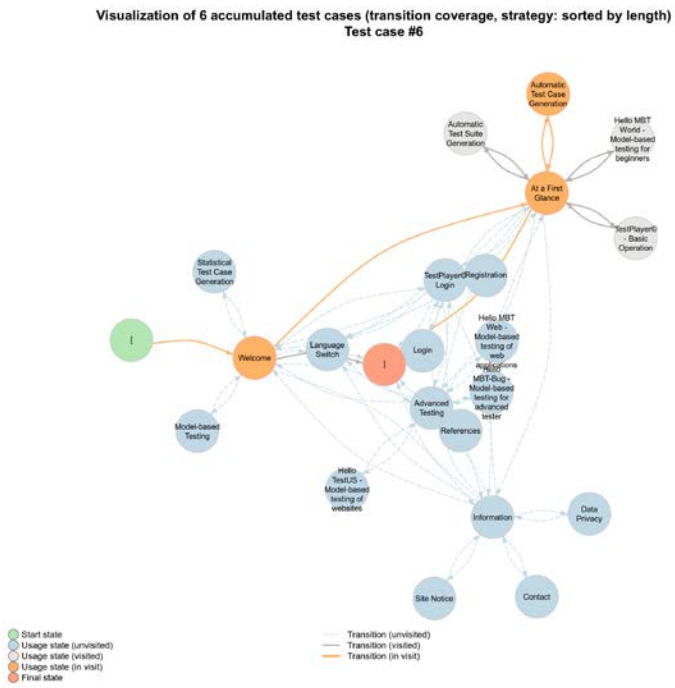


Fig. 15: Simulated annealing layout for test case visualization.

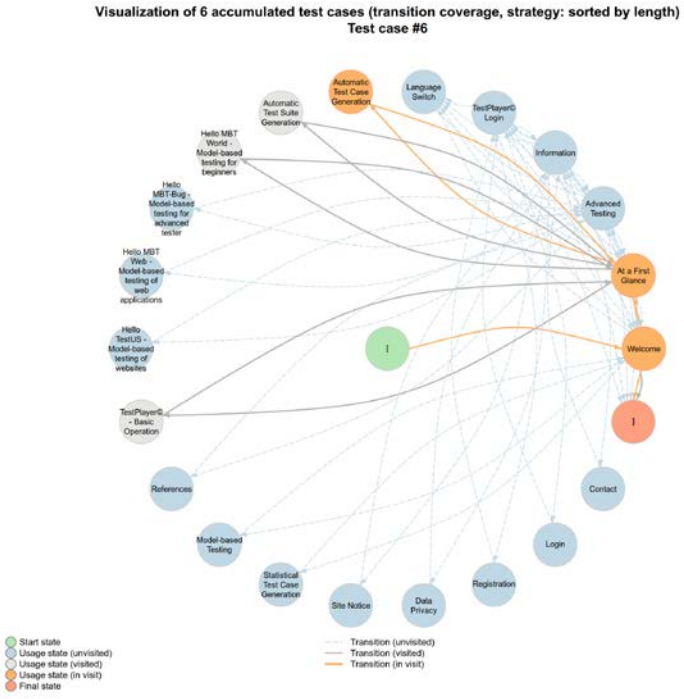


Fig. 16: Star shaping layout for test case visualization.

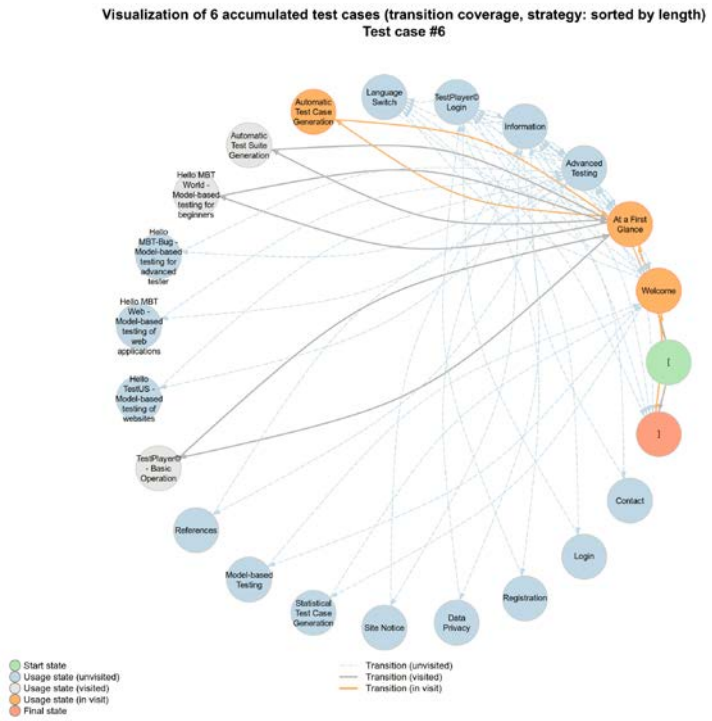


Fig. 17: Circle shaping layout for test case visualization.

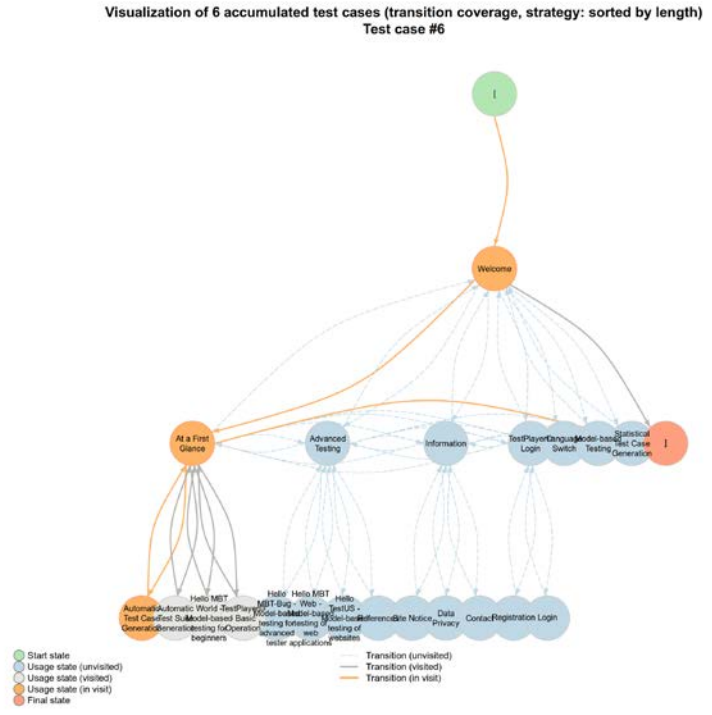


Fig. 18: Tree shaping layout for test case visualization.

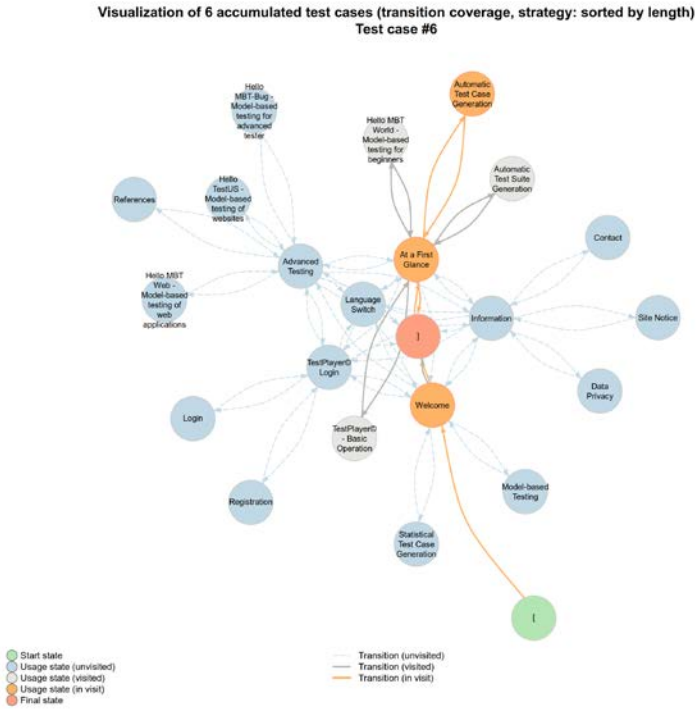


Fig. 19: Force-directed layout for test case visualization

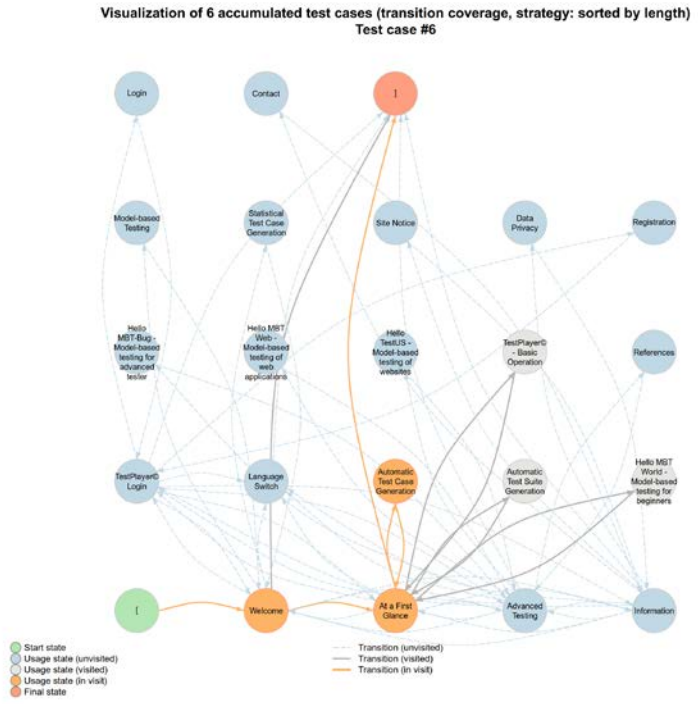


Fig. 20: Rectangular grid layout for test case visualization

Conclusion and Lessons learned

In this whitepaper, we have shown that model-based testing based on statistical usage models provides an excellent enhancement compared to conventional deterministic approaches for automated testing.

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 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 and experience in recent years can be summarized as follows:

- Model-based testing is an innovative test approach to improve the effectiveness and efficiency of the testing process.
- A model-based tester uses models to control test design and analysis and takes advantage of models for other test activities such as test case generation and test report generation.
- 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.
- 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 [6].
- 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.
- The TestPlayer© is independent of special application domains and can be used in a variety of ways.

References

[1] W. Dulz. **A Comfortable TestPlayer© for Analyzing Statistical Usage Testing Strategies**. Proceedings of the 6th ICSE/ACM Workshop on Automation of Software Test (AST '11), 2011.

[2] W. Dulz. **Model-based Strategies for Reducing the Complexity of Statistically Generated Test Suites**. Proceedings of the 2013 Software Quality Days, 2013.

[3] W. Dulz. **Model-based Testing of Multilingual Websites and Web Applications**. Proceedings of the 13th International Conference on Software Technologies (ICSOFT), 2018.

[4] W. Dulz. **Theory and Practice in Testing using Statistical Usage Models**. Tutorial at the 13th International Conference on Software Technologies (ICSOFT), 2018.

[5] Selenium HQ Browser Automation.

[6] W. Dulz, S. Holpp and R. German. **A Polyhedron Approach to Calculate Probability Distributions for Markov Chain Usage Models**. Electronic Notes in Theoretical Computer Science, Volume 264, Issue 3, December 2010, Pages 19-35.

[7] A. Narayanan. **Test Automation ROI Calculator**. White Paper, Aspire Systems.

[8] S. Münch, P. Brandstetter, K. Clevermann, O. Kieckhoefel, and E. R. Schäfer. **The Return on Investment (ROI) of Test Automation. Pharmaceutical Engineering**. Pharmaceutical Engineering, Vol. 32/No. 4, 2012.

[9] W. Dulz. **On-the-Fly Testing by Using an Executable TTCN-3 Markov Chain Usage Model**. Evaluation of Novel Approaches to Software Engineering. 3rd and 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Revised Selected Papers, p. 17-30, 2008.

[10] S. Siegl, W. Dulz, R. German, G. Kiffe. **Model-Driven Testing based on Markov Chain Usage Models in the Automotive Domain**. Proceedings of the 12th European Workshop on Dependable Computing (EWDC), 2009.

[11] A. Djanatliev, W. Dulz, R. German, and V. Schneider. **VeriTAS - A Versatile Modeling Environment for Test-driven Agile Simulation**. Proceedings of the 2011 Winter Simulation Conference (WSC), 2011.