Automated Testing at the GUI level

Hands-on do it yourself manual

2019

The TESTAR team

Last updated: 14/11/2019 at 11:39.

TESTAR research is funded by the following projects.

ITEA3 TESTOMAT Project will support software teams to strike the right balance by increasing the development speed without sacrificing quality. The project will ultimately result in a Test Automation Improvement Model, which will define key improvement areas in test automation.

https://www.testomatproject.eu/

DECODER builds an Integrated Development Environment (IDE) that combines information from different sources through formal and semi-formal models to deliver software project intelligence to shorten the learning curve of software programmers and maintainers and increase their productivity.

https://www.decoder-project.eu/

"Extended Reality" (XR) systems are advanced interactive systems such as Virtual Reality (VR) and Augmented Reality (AR) systems. IV4XR aims to build a novel verification and validation technology for XR systems based on techniques from AI to provide learning and reasoning over a virtual world.

https://iv4xr-project.eu/

IVVES will systematically develop Artificial Intelligence approaches for robust and comprehensive, industrial-grade V&V of "embedded AI", i.e. machine-learning for control of complex, mission-critical evolving systems and services covering the major industrial domains in Europe.

https://ivves.weebly.com/

Contents

1 What is GUI Testing? 5

2 We start with a simple buggy SUT 6

3 The Virtual Machine 6

4 Downloading and installing TESTAR 7

5 Hands-on step by step 9

5.1 Manually test the SUT 9

5.2 Starting up TESTAR 9

5.3 Introduction to TESTAR GUI 10

5.4 The SPY Mode 12

5.5 The GENERATE mode 14

5.6 Test Reports Structure 15

5.7 The REPLAY mode 18

5.8 The VIEW mode 18

5.9 Other ways of looking at the test results 19

5.10 RECORD mode 20

5.11 Design a Test Oracle 20

5.12 Adjust TESTAR's Behavior 21

5.13 The test.settings file 22

5.14 Listening to processes 24

5.14.1 Suspicious Process Output oracles 25

5.14.2 Save process execution logs 25

5.15 Connecting with a SUT: the SUT connectors 26
5.16 Execute TESTAR from the command line 27
5.17 Play with the Tool and some other applications 28
5.18 Testing web applications 29
5.19 Editing the protocol 29
5.19.1 Editing the protocol to logon 29
5.19.2 Editing the protocol to add an oracle 32
5.20 Testing Java Swing applications 33
5.21 TESTAR's generated statemodel 35
5.21.1 Using the graph database 36

6 WebDriver **38**

6.1 Installing the Selenium Webdriver 38
6.2 Start TESTAR 40
6.3 Configure webdriver path 41
6.4 Discover webdriver features 43
6.4.1 The whitelisting of domains 43
6.4.2 The blacklisting of extensions 44
6.4.3 Add clickable elements 44
6.4.4 Blocking overlays and password protection 46

A Keyboard shortcuts **47**

B Directories **47**

C Test settings **48**

1 What is GUI Testing?

GUI Testing is a testing technique where one tests the System Under Test (SUT) solely through its Graphical User Interface (GUI). The only way to find errors is to thoroughly observe the status of the GUI after each step of the test. This type of testing is usually still carried out manually, where a tester just follows a previously written "test case" and verifies whether the application responds to all inputs as expected.

On the one hand GUI testing is a relatively straightforward process, since one does not need to read or test the source code of the SUT. On the other hand it is quite laborious, time consuming and, well... boring...

Approaches for automation of GUI testing are for example Capture & Replay or visual GUI testing. These approaches use test scripts to automate the steps of each test case. These type of tests, whether manual or automated through scripts, usually test the sunny day use cases and mostly follow the happy paths. But what about the less sunny test scenarios? Those scenarios executed by the rogue user?

In this manual we will look at automation of GUI testing using a tool called *TESTAR* [1]. TESTAR is a tool that automatically tests also the less sunny use cases through random and monkey tests. It hence complements the other approaches:

TESTAR is an open source tool for automated testing through GUI. It automatically interacts with the SUT by clicking and typing on the con-

[1] TESTAR is originally a result of FITTEST, a European project that run from 2010 till 2013. More info `www.testar.org`

trols of the GUI. It is able to recognize abnormal SUT behaviour, such as crashing or freezing. Moreover, you can add application specific test oracles that are able to detect domain specific faults. TESTAR reports test sequences that lead to failure.

2 We start with a simple buggy SUT

To get a first impression of the challenges associated with GUI testing, you will be using a given System Under Test (SUT) that consists of a simple Calculator with several failures. Your task is to setup a test which finds the failures within this SUT. To achieve this, you will use TESTAR which automatically explores GUI-based applications and can – once configured correctly – detect specific types of errors. Your goal is to find as many errors as possible and reproduce them using TESTAR.

3 The Virtual Machine

For this hands-on we created a virtual machine image with all the required software already installed. To run the image you need to have VirtualBox version 5.2.18 or later. If you do not have VirtualBox 5.2.18 on your machine you can download it here: `https://download.virtualbox.org/virtualbox/5.2.18/` .

If you have a newer version ($>$ 5.2.18) of VirtualBox installed, you might need to update the "VBox Guest Additions" on the image to the same version as your VirtualBox version. Please refer to the VirtualBox manual on how to do this.

Minimal resource requirements for the VirtualBox Host:

- VirtualBox version $\geq$ 5.2.18
- Recent Dual core CPU or better
- 8 GB Memory
- about 80 GB free diskspace

The hands-on OVA image can be downloaded from the Testar website (`https://testar.org/images/`<*image_name*>). You will be given

<*image_name*> during the training-sessions. If you are doing the hands-on on our own, please send an email to `info@testar.org` to obtain the latest image or check from `https://testar.org/download/` and look for the latest OVA image.

The OVA image is about 22GB downloaded.

☞ **HANDSON**: Import the downloaded OVA image in VirtualBox via "File" → "Import Appliance".

Note: After importing the OVA image about 50GB is used by the virtual machine. The disk-usage will grow with usage and is maximized at 200GB, with normal usage it should not grow beyond 60GB during this Hands-on.

On Windows 10 you can use:

username: "`testar`"

password: "`testar`".

☞ **HANDSON**: Startup the virtual machine in VirtualBox and try to login as the "testar" user.

4 Downloading and installing TESTAR

You will have to download the testar_<version number>.zip from the TESTAR website and unpack the zip file into the `C:\` directory with the instructions below.

- Go to `https://testar.org/download/`
- Download the TESTAR zip distribution created for this Handson.
- Go to the windows downloads directory and right-click on the just downloaded testar_<version number>.zip file. Select the "Extract ALL" option in the menu. (see Figure 1)
- Select for the destination directory `C:\` . Click the "Extract" button, this will extract the testar_<version number>.zip file, the installation can be found in `C:\testar\` (see Figure 2)

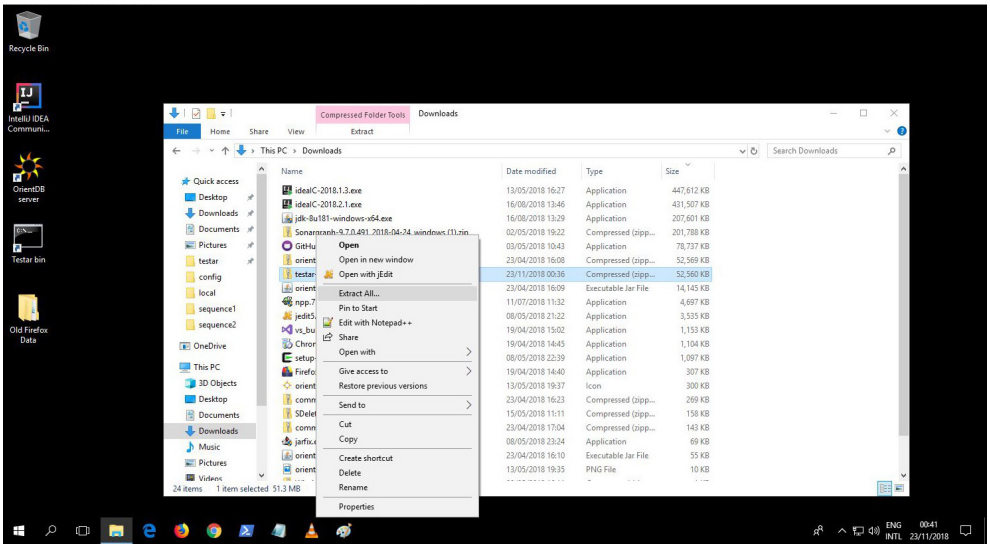

Figure 1: Right click and select "Extract all"

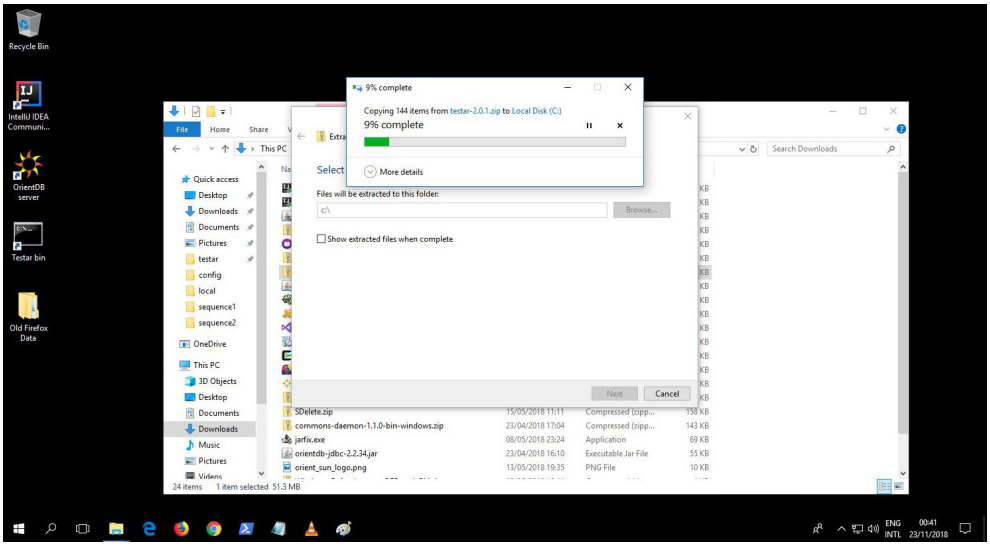

Figure 2: Unzip testar_<version number>.zip file in windows

5 Hands-on step by step

5.1 Manually test the SUT

Before using TESTAR, test the SUT manually, to get an impression of potential failures. You can find the Calculator in `suts\Calculator.jar` folder, as part of the TESTAR binaries you just extracted. You can either double click on the `Calculator.jar` file or in the command prompt type `java -jar suts\Calculator.jar`.

HANDSON: How many and what type of failures can you find?

5.2 Starting up TESTAR

Now let us start up TESTAR to see what it can do. In the `bin` directory of the unpacked `testar`-directory there is a `testar.bat`.

HANDSON: Open the windows prompt that is on the desktop and type `testar.bat`. On Enter the GUI of TESTAR starts up.

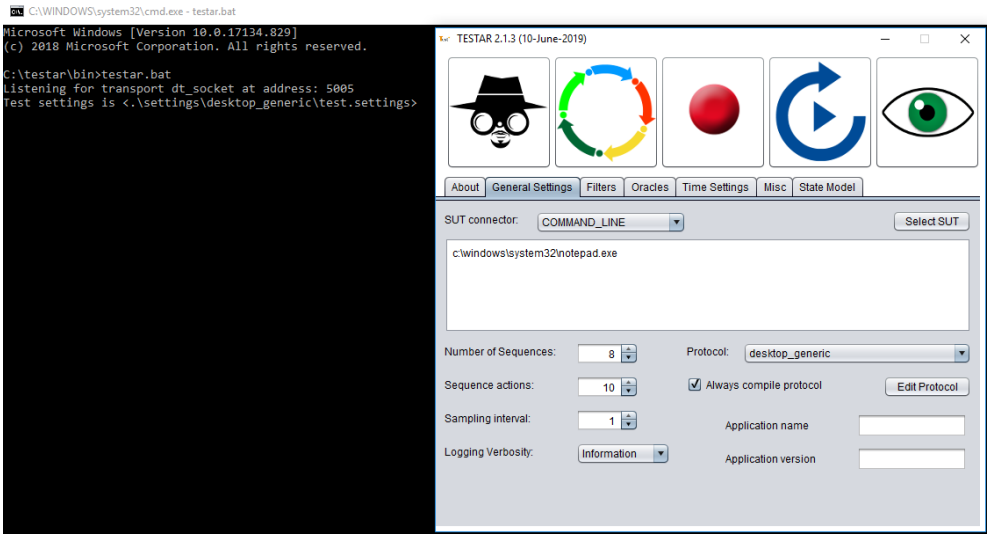

Figure 3: Starting up TESTAR

NOTE: If your `JAVA_HOME` path does not point correctly to the installation path for the Java Development Kit you will see a

`java.lang.RuntimeException: JDK required (running inside of JRE)`

To set your `JAVA_HOME` look for example on stackoverflow[2] for the instructions for your type of machine.

NOTE: sometimes, if the configuration is not complete, something like Figure 14 from Section 13 is shown first. Then, at this point, you should select the *desktop_generic* and upon pressing OK, the GUI from TESTAR like in Figure 3 starts up.

5.3 Introduction to TESTAR GUI

In the following sections we are going to start from the surface and continue to dive deeper into the mysteries of TESTAR, on the journey to become an advanced user of TESTAR :-)

At the moment, please don't click anywhere on the TESTAR tool, first you should read this short introduction about the impressions that TESTAR GUI offers to users.

The graphical user interface of TESTAR is a representation of the different execution modes that this tool can offer to inspect or test the SUTs / applications. As you can see in Figure 4, we are going to start with introducing the 5 different "sections" of the General functionality (colored accordingly in the Figure below).

TESTAR execution modes (Yellow) :

These icons represents the different modes with which you can inspect or test the desired application/SUT. In order these are:

Spy (Check what TESTAR can recognize and obtain from the SUT)

Generate (Run tests automatically)

Record (User can execute actions manually)

REPLAY (Re-execute an existing test sequence)

VIEW (See what TESTAR has executed)

TESTAR settings customization (Purple) :

We will navigate around these settings panels to customize time options, Oracles to find errors, Filter widgets or actions we don't want to execute, etc.

[2] `https://stackoverflow.com/questions/2619584/how-to-set-java-home-on-windows-7`

TESTAR SUT connector (Green) :
Here we will define which application or SUT (System Under Test) we want to test, and how we want to connect with it.

TESTAR protocol (Red) :
At this moment make sure you are using the *desktop_generic* protocol, this functionality can be confusing. Different technologies or software requires different ways to connect, also different SUTs could require specific customization of settings or "flow" executions (see Figure 6).

TESTAR tests iterations (Blue) :
These are settings for **Generate** mode, to choose how many iterations you want to execute
`Start SUT - Execute X actions - Close SUT - Start again`.

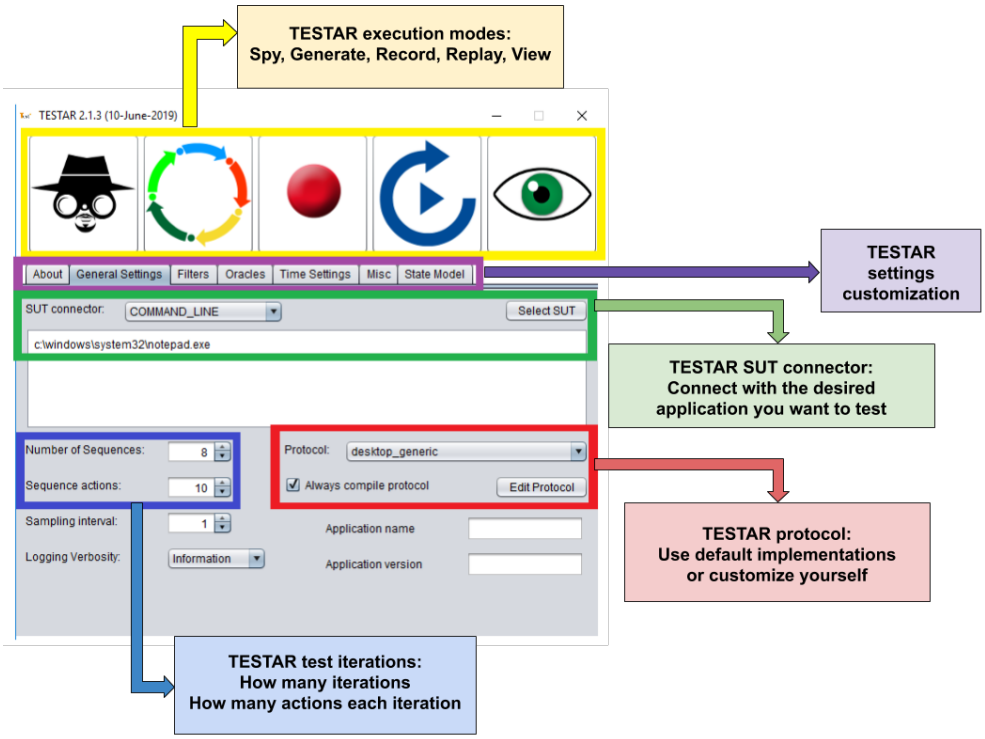

Figure 4: Schema of GUI functionality

Basically the GUI of TESTAR is a dialog for configuring the values that are present in the test.settings file. These settings define details for TESTAR on how to test a specific SUT.

For example, in the "General Settings"-tab you can see a SUT connector setting that defines the way TESTAR connects to the SUT. The default is to connect through `COMMAND_LINE`. In the text field you can type the following command to indicate to TESTAR that you want to connect to the Calculator as a Java jar package:

```
java -jar suts\Calculator.jar
```

TESTAR has various types of protocols to "connect" or interact with diverse desktop or web applications, according to their technology, and you can write new ones for your applications.

From the drop-down menu "Protocol" we can select the protocol we want to use for testing. Since the Java Calculator has been created with Java AWT GUI technology, in TESTAR it's in the category of generic desktop applications, let us select: *desktop_generic*.

Other protocols like *web_generic* for web applications (see section 5.18) or *desktop_swingset2* for Java Swing applications (see section 5.20) will be used later.

Now let us start by clicking on the *SPY* button (the one with the magnifying class). This one enables us to spy the buttons and other widgets of the SUT and see all the information that TESTAR is able to extract. Hover over the different parts of the GUI and look for yourself.

5.4 The SPY Mode

The SPY mode allows you to inspect the widget controls of the GUI. In the spy mode you can:

- See what actions TESTAR is able to extract and choose from. The green dots represent the available widgets a user can click on in that specific state of the GUI.
- Hover over an element to show the properties and information of that element or widget.
- Press Shift + ⇑ to show extended information and properties while hovering over an element. This way you can find more details, for example about the titles of the elements. To go back to less information just press Shift + ⇑ again.

To leave SPY mode and return to TESTAR start-up dialog, press Shift + ↓, or close the system you were spying.

Figure 5 gives an overview of how to go from mode to mode and back to TESTAR. A summary of the shortcuts in Appendix A. It is a good idea to become familiar with this Mode and its shortcuts.

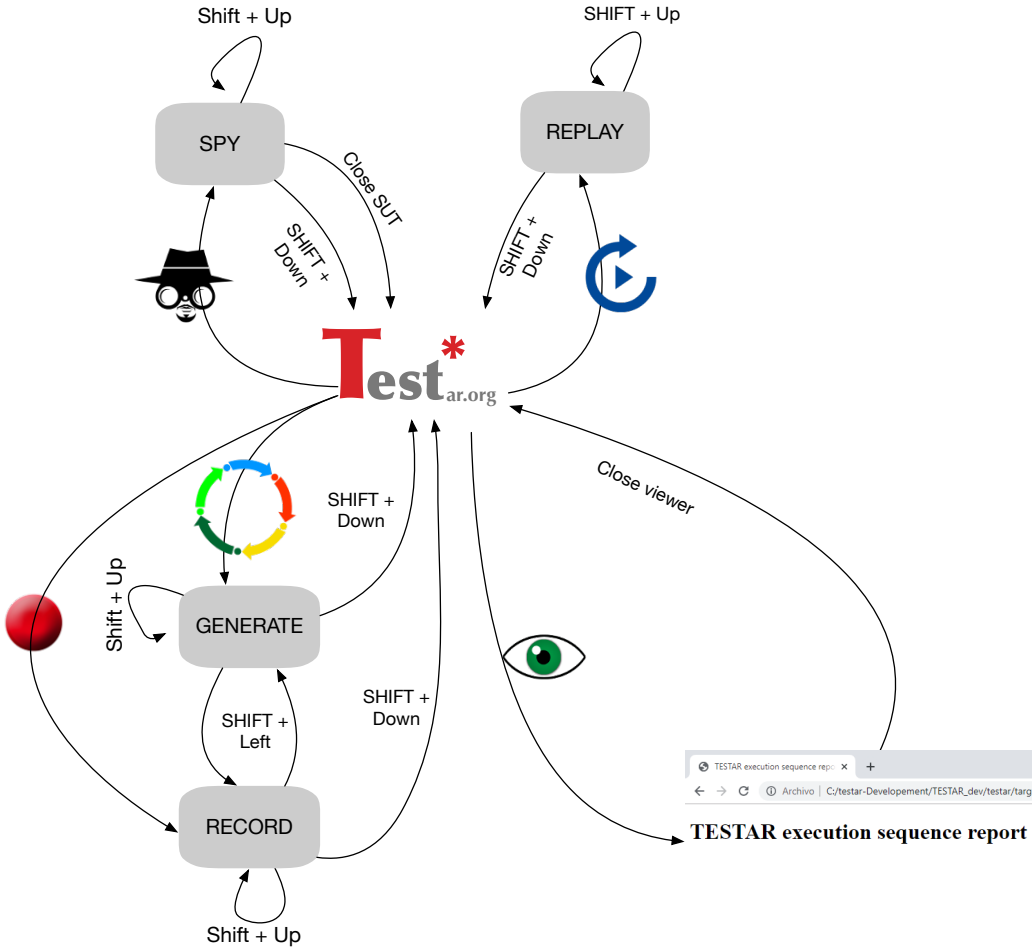

Figure 5: TESTAR Modes

5.5 The GENERATE mode

In this mode, the TESTAR tool carries out automated testing following the test cycle depicted in Figure 6.

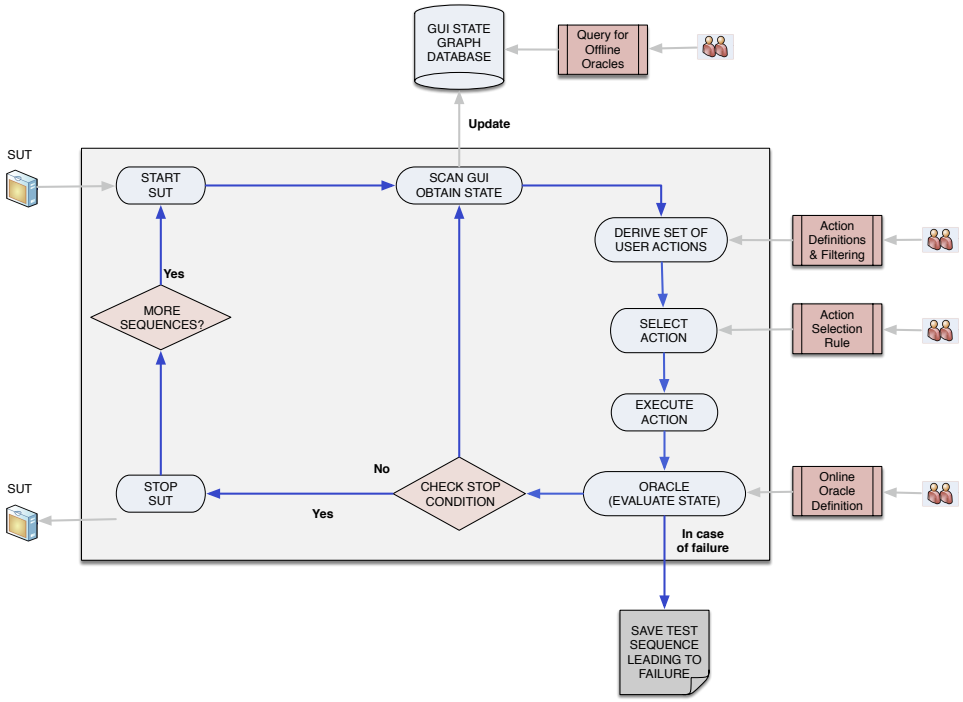

Figure 6: TESTAR test cycle

Basically, it derives a set of possible actions for the current state that the GUI of the SUT is in. Then, it automatically selects and executes an action from this set which makes the SUT go to a new GUI state. This new state is evaluated with the available oracles. If no failure is found, again a set of possible actions for the new state is derived, one action is selected and executed, etc. This continues until a failure has been found or until a stopping criteria is reached. With the right test setup all you'll need to do is to wait for your tests to finish.

The default behaviour includes random selection of actions and implicit oracles for the detection of the violation of general-purpose system requirements:

- the SUT should not crash,

- the SUT should not find itself in an unresponsive state (freeze), and
- the UI state should not contain any widget with suspicious titles like *error*, *problem*, *exception*, etc.

In the "General Settings"-Tab you can configure the number of test sequences you want to test and the length (i.e. number of actions) of these sequences. For now, since we are learning the tool, let us not put too many. Say 3 sequences of 10 actions each.

Now we are ready to do some automated testing. Click on the **Generate** button to start running tests as specified by the test setup. These are monkey test. TESTAR can *see* the controls of the SUT's UI and can automatically detect possible actions. It randomly selects and executes these actions.

Like in the *SPY* mode, you will be able to toggle on/off the visualization of the UI actions by pressing Shift + ↑. The following color codes are applied for visualization during the test:

- **green dots** for UI actions that TESTAR can detect and execute, and
- **red dot** for the currently selected UI action being executed.

5.6 Test Reports Structure

When the tests have finished, we can inspect the results of our tests in the directory:

```
\testar\bin\output\
```

The folders in this directory represent the different test runs (see Figure 7). The names of the folders have the format `timestamp_SUTname`. By default, TESTAR will try to obtain the name of the SUT application (i.e. `SUTname`) by reading the path of the executed application. However, sometimes, when more clear names are desired, it can be better to configure the name in the "Application name" field that we can see in Figure 3.

Inside the different folders you can find all reports and sequences that TESTAR generated during the test run the folder represents. The reports and sequences results can be divided in up to 8 directories as you can see in Figure 8.

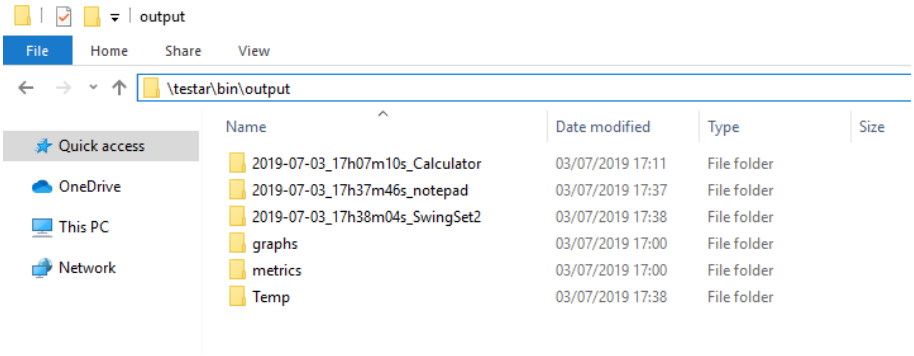

Figure 7: TESTAR output directory

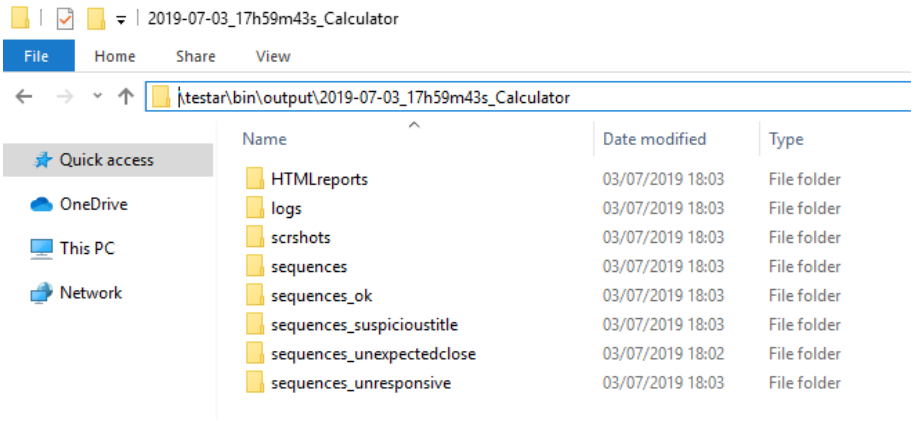

Figure 8: TESTAR specific run directory

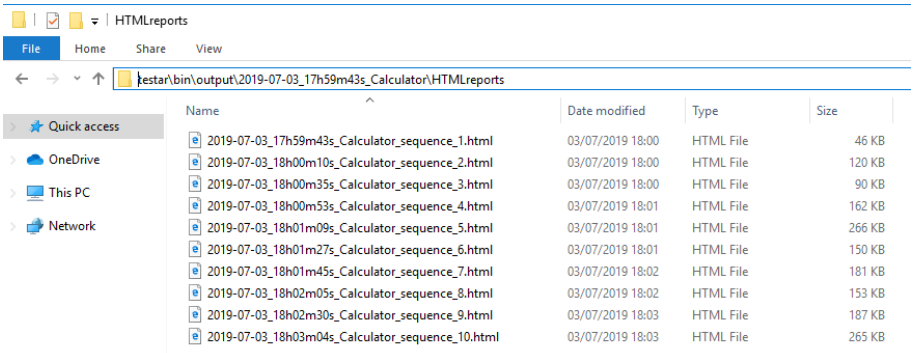

Figure 9: TESTAR HTMLreports directory

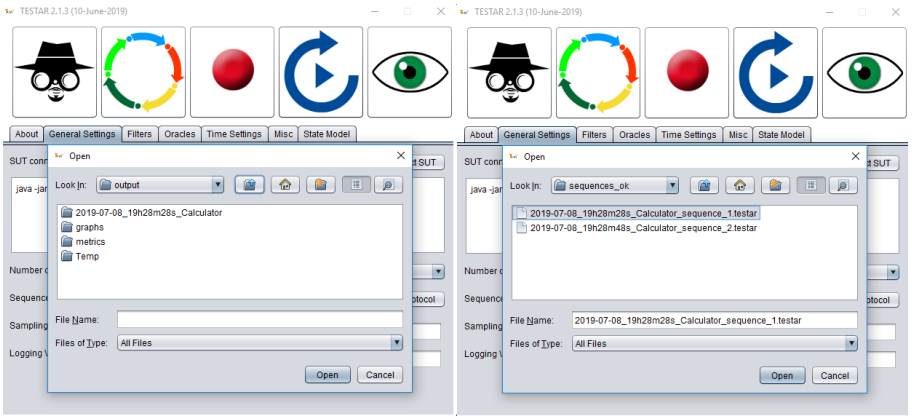

Figure 10: TESTAR select sequence pop up window

HTMLreports This folder contains HTML reports detailing the different states, widgets and actions that were found and executed by TESTAR during the run. These can be opened in the VIEW mode. An example is in Figure 9 were we see the resulting reports of a TESTAR run that contained 10 sequences.

logs Details about the executed actions and their attached Widget in plain test format.

scrshots This folder contains the screenshots of the GUI state taken by TESTAR throughout the execution of actions.

sequences This folder contains all the sequences that have been executed by TESTAR. The files in these directories are binary files (with the extension `.testar`) and can only be read by TESTAR in REPLAY mode. We will look at this mode in the next section.

sequences_ok This folder contains all those sequences from the folder **sequences** that have not resulted in failure.

sequences_suspicioustitle This folder contains all those sequences from the folder **sequences** that have triggered the suspicious title oracle and resulted in failure.

sequences_unexepectedclose This folder contains all those sequences from the folder **sequences** that resulted in a crash.

sequences_unresponsive This folder contains all those sequences from the folder **sequences** that resulted in the SUT being unresponsive.

5.7 The REPLAY mode

When you click the REPLAY button then a window pops up. This file explorer window allows you to select the sequence that you want to replay or view (Figure 10).

The REPLAY mode can replay sequence files that end with the extension `.testar`. These files.testar are those that can be found in the directories of the different sequences seen in Figure 8.

REPLAY will play the saved sequence again, repeating the same actions executed previously. This can be used to verify that a correct sequence (`sequences_ok`) still running without fails after the release of a new version, or to check that the found failures are solved with a fix release.

5.8 The VIEW mode

The execution of the VIEW mode opens the same windows pop up (Figure 10) that allows you select the sequence you want to view.

In this mode you can chose an HTML report of the sequence or a plain text `.log` file. TESTAR will use the default system applications defined as a visualization tool (Microsoft Edge and Notepad into the Virtual Machine) to open the selected files.

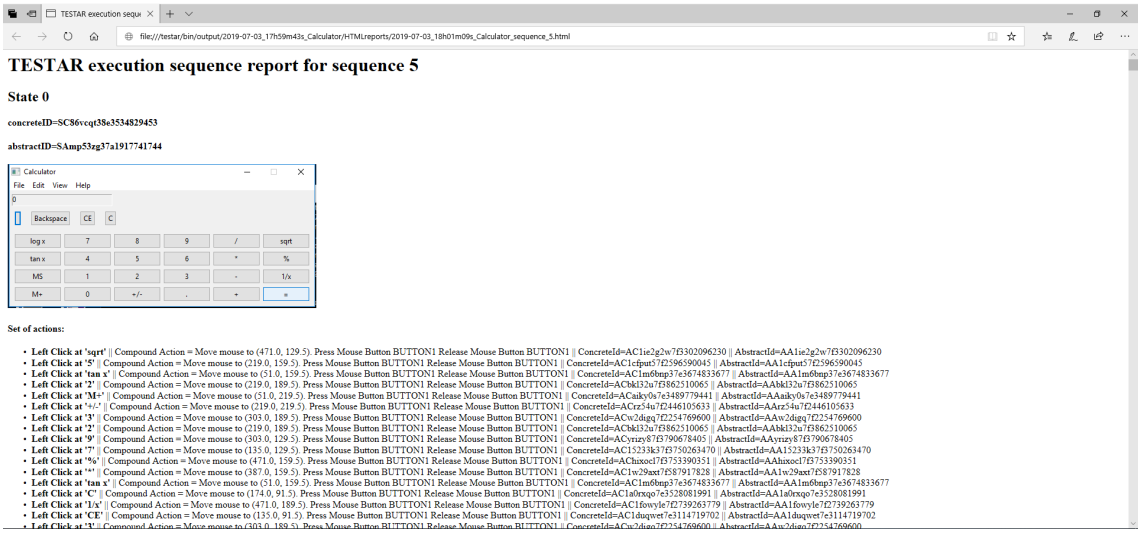

Figure 11: TESTAR HTML report (State information)

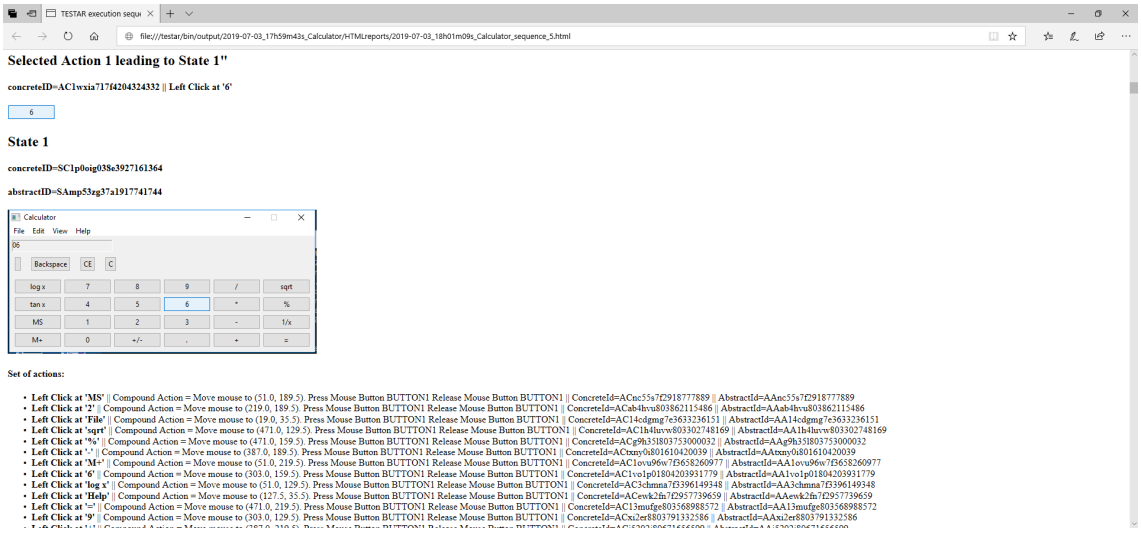

Figure 12: TESTAR HTML report (Action information)

For example, in Figure 11 and 12, you can see the HTML report of of a test sequence executed with the buggy calculator, that contains the obtained information by TESTAR about the founded States, Widgets and available Actions.

HANDSON: Start up TESTAR and click on the VIEW button on the top right side of the TESTAR GUI. Select one of your previously executed test sequences.

HANDSON: Configure TESTAR to generate a test sequence with a maximum of 30 actions and then replay it. To replay it you can use the REPLAY button and specify the path to the sequence.

5.9 Other ways of looking at the test results

TESTAR also generates a summary log report that contains the timestamp, directory, file and result of all the executed sequences. (**NOTE**: this file will be created from the first run on and extended each run).

`\testar\bin\index.log`

And a set of detailed logs to investigate what happened during the execution of a specific sequence. You can find these in:

`\testar\bin\output\timestamp_SUTname\logs`

You can open them with for example `Notepad++`, `WordPad` or using the VIEW mode to inspect them. There are a lot of incomprehensible identifiers there which will come in handy later on for root cause analysis. But you can also see all the steps the sequence consists of, and the reason why the passed or failed.

These logs, especially the `index.log`, could be useful to integrate TESTAR tool and the results of the sequences into a CI pipeline.

5.10 RECORD mode

In **Record**-mode you will be able to take over TESTAR's test control and manually perform some actions during a test. This is particularly interesting if you want to force the test to move to a concrete part of the UI and/or if you want to move the test out of the current part of the UI.

You can go from **Generate** to **Record** mode and back with the keyboard shortcut Shift + ←.

☞ **HANDSON**: Run some tests. Try to interact during the tests at any time by going to the **Record** mode. You can stop by pressing shortcut (Shift + ↓) to go back to the TESTAR start-up dialog.

5.11 Design a Test Oracle

The default behavior of TESTAR can find only certain types of failures. In order to detect a wider variety of failures, TESTAR allows the user to define application specific *oracles*. The oracle tells whether a specific GUI state is correct, faulty or suspicious. TESTAR comes with default implicit oracles that can detect the violation of the general-purpose system requirements mentioned above.

While you were playing with the SUT a bit, you probably detected a few failures on your own, such as dialogs with exception messages. To detect such failures, one can simply scan the current GUI state for specific words, such as "exception" or "error" or "NullPointerException", etc. The "Oracle"-Tab has a field "Suspicious Titles" in which you can write Java

Regular Expressions[3][4]. TESTAR will use these expressions after the execution of each action in order to find potential matches with the titles of the widgets. For example:

```
.*[Ff]aultystring.*|SomeOtherFaultyString
```

This expression will make TESTAR look for the string "Faultystring" (upper- or lowercase F) anywhere on the screen in any position as well as for the exact match "SomeOtherFaultyString". If TESTAR encounters such a string, it will verdict a suspicious output and save the corresponding sequence under:

```
\testar\bin\output\timestamp_SUTname\sequences_suspicioustitles\
```

☞ **HANDSON**: Write regular expressions for the failures that you have encountered in the SUT. Run some tests. Did you find any failures with your oracles? Replay a test sequence that found a bug.

5.12 Adjust TESTAR's Behavior

You might have observed, that from time to time, TESTAR executes "undesirable" actions. For example, actions that minimizes the SUT or even terminates the SUT. This is not optimal for the testing process. Even worse: If you observed the tool's output you might have noticed that it detects a "crash" whenever it closes the main window. Obviously, the tool does not know that closing the main window terminates the application. In the "Filters"-Tab you can find an input field that allows you to filter actions by defining a regular expression. The field is similar to the "Suspicious Titles" in the "Oracles"-Tab. TESTAR will ignore all actions that exercise control elements whose title matches the given regular expression. For example:

```
.*Backspace.*|.*CE.*|.*View.*
```

This expression will ignore clicks to all control elements whose titles contain the given strings.

☞ **HANDSON**: Configure TESTAR such that it does not close or minimize the window anymore. In addition disallow clicks to the "Open File" menu item, to prevent TESTAR to go wild on the operating system's files. Use the Spy Mode to see whether your action filters have an effect!

[3] See for example `http://www.vogella.com/tutorials/JavaRegularExpressions/article.html`

[4] `https://zeroturnaround.com/rebellabs/java-regular-expressions-cheat-sheet/`

You can also filter actions while being in SPY mode, by using the *clickfilter*-feature. CAPS_LOCK toggles the clickfilter feature that enables you to filter actions by clicking on them in Spy mode. This comes in very handy when setting up your tests. Once this feature is enabled, you can just hover over the widget and press Ctrl to filter the actions on this widget from being selected during testing (you don't have to press mouse button on the widget). You can undo the filtering by pressing Shift + Ctrl while hovering over a filtered widget. If the action filter you specified with a regular expression in the "Filters"-Tab was "too efficient", you can unfilter a specific action this way. You can also filter several actions at once by a dragging a square around the widgets while pressing Ctrl . Filtered actions will be stored in the file `protocol_filter.xml` that you can find in the current TESTAR's protocol folder (`\testar\bin\settings\desktop_generic`).

☞ **HANDSON**: Start the calculator in SPY mode and press Shift + 1 to visualize all the green dots (those are the available, unfiltered actions). Then enable the clickfilter feature by pressing CAPS_LOCK and filter out some actions to see the effect of green dots disappearing. You can also check the `protocol_filter.xml`.

5.13 The test.settings file

As indicated before, the start-up dialog of TESTAR enables us to configure the values that are present in the test.settings file.

We have provided a predefined set of test.settings for generic desktop and web applications. You can find them under the *settings* folder in TESTAR's bin folder (`\testar\bin`) Each test settings configuration is stored inside a unique subfolder (e.g. *desktop_generic*), which contains:

1. a Java source file (e.g. `Protocol_desktop_generic.java`) with the programmable test protocol (more about that later in Section 5.19).
2. a test.settings file, which contains a list of test properties.
3. a `protocol_filter.xml` file, if there is filtered actions as we have commented previously.

In the settings directory we can see a file ending with the extension `.sse`. This file is used to indicate from which folder TESTAR will choose the settings. This protocol selection can be changed by editing the file.sse

```
##################################################################
# TESTAR mode
#
# Set the mode you want TESTAR to start in: Spy, Generate, Replay
##################################################################

Mode = Spy

##################################################################
# Connect to the System Under Test (SUT)
#
# SUTCONNECTOR = COMMAND_LINE, SUTConnectorValue property must
# be a commandline that starts the SUT. It should work when
# typed into a command prompt (e.g. java -jar SUTs/calc.jar). For
# web applications, follow this format: web_browser_path SUT_URL.
#
# SUTCONNECTOR = SUT_WINDOW_TITLE, then SUTConnectorValue property
# must be the title displayed in the SUT' main window. The SUT
# must be manually started and closed.
#
# SUTCONNECTOR = SUT_PROCESS_NAME: SUTConnectorValue property must
# be the process name of the SUT. The SUT must be manually started
# and closed.
##################################################################

SUTConnector = COMMAND_LINE
SUTConnectorValue = java -jar "\local\suts\Calculator.jar"

##################################################################
# Sequences
#
# Number of sequences and the length of these sequences
##################################################################

Sequences = 6
SequenceLength = 100

##################################################################
# Oracles based on suspicious titles
#
# Regular expression
##################################################################

SuspiciousTitles = .[eE]rror.*|.*[eE]xception.

##################################################################
# Actionfilter
#
# Regular expression. More filters can be added in Spy mode,
# these will be added to the protocol_filter.xml file.
##################################################################

ClickFilter = .[cC]lose.*|.*[fF]ile.*|.*Minimize.*|.*[mM]inimizar.
```

Figure 13: Part of the test.settings file

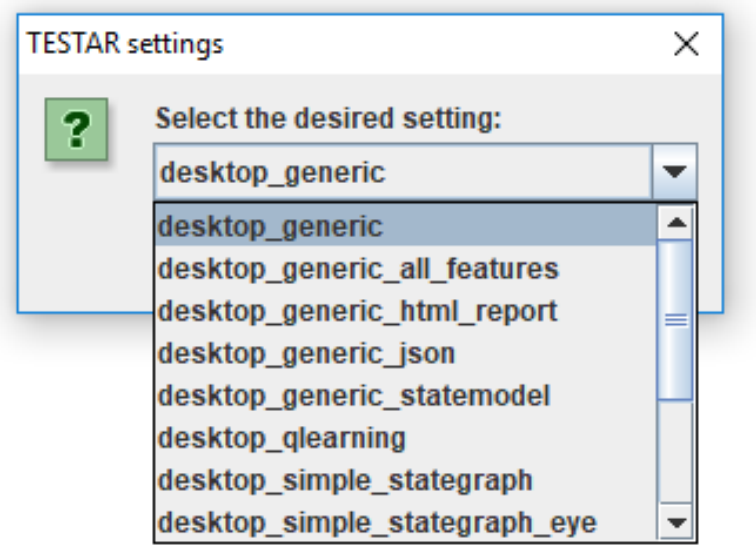

Figure 14: TESTAR when started without an .sse file

name directly or through the TESTAR user interface, selecting the desired protocol from the corresponding dropdown-menu in the "General Settings"-Tab.

If the `.sse` is not present in the *settings* folder, a dialogue like the one in Figure 14 will start up to ask you to select the protocol that you would like.

If you want, you can also edit the files directly. For now, to illustrate we refer to Figure 13, where you can see part of the test.settings file for the calculator application. More settings are explained in detail in Appendix C.

☞ **HANDSON**: Change the name of the `.sse` file and start up TESTAR again to see the effect. What if you delete the `.sse` file?

5.14 Listening to processes

TESTAR can also interact with the process of desktop applications, listening to the buffers of its process. This is useful because it allows the tester to define oracles enable to find *suspicious output* coming from the processes, similar in the way that it looks for *suspicious titles* in the different widgets that make up the interface (Section 5.11). Moreover, we can store the output of the processes in logs for later inspection to find anomalies.

In order to use these features, we need to enable the process listeners on the test.settings file:

```
ProcessListenerEnabled = true
```

5.14.1 Suspicious Process Output oracles

To detect Suspicious Process Output, the tester has to specify the oracles that define the suspicious outputs in the same way seen the Section 5.11, we can specify them in the test.settings file:

```
#################################################################
# Oracles based on Suspicious Outputs detected by Process Listeners
#
# Requires ProcessListenerEnabled
# (Only available for desktop applications through COMMAND_LINE)
#
# Regular expression SuspiciousProcessOutput contains the specification
# of what is considered to be suspicious output.
#################################################################

SuspiciousProcessOutput = .*[eE]rror.*|.*[eE]xcep[ct]i[o?]n.*
```

Currently these oracles based on process listeners are only enabled for desktop applications executed from the `COMMAND_LINE` option. When an oracle finds a suspicious title, execution of the test is stopped.

5.14.2 Save process execution logs

Enabling the process listeners means that we can save, filter and store output during the test execution. You can configure this in test.settings as follows:

```
#################################################################
# Process Logs
#
# Requires ProcessListenerEnabled
# (Only available for desktop applications through COMMAND_LINE)
#
# Allow TESTAR to store execution logs coming from the processes.
# You can use the regular expression ProcessLogs below to filter
# the logs. Use .*.* if you want to store all the outputs of the
# process.
#################################################################

ProcessLogs = .*.*
```

☞ **HANDSON**: Start the calculator in the Spy mode, filter all widgets except "View" and the menu item "Digital Grouping". Now run the Generate mode. What happened?

The calculator closed unexpectedly. This type of failure was recognized by the default oracles built into TESTAR, and therefore the sequence will be stored in the file:

`\testar\bin\output\timestamp_SUTname\sequences_unexpectedclose\`

With the process listener disabled, we only know that the SUT has closed unexpectedly or that something froze the application. However, when we activate the process listener we can obtain a lot more information about the failure detected by TESTAR.

☞ **HANDSON**: Run TESTAR again, but now in the Oracles tab, check the tickbox `Enable Process Listener` and add `.*Exception.*` as a Suspicious Process Output. Run the calculator again with the Generate mode.

Now the HTML report will include a message to indicate that the Verdict comes from the Process Listener, and the output process information it will be stored according to its sequence in the new folder:

`\testar\bin\output\timestamp_SUTname\logs\processListener\`

5.15 Connecting with a SUT: the SUT connectors

Until now we have connected to the SUT through the `COMMAND_LINE`. In the "General Settings"-Tab, that option was selected from the drop-down menu. In the text field we indicated that the `COMMAND_LINE`-command to connect to the SUT was: `java -jar \local\suts\Calculator.jar`.

However, if a Java application has to be executed using the .exe file, it is better to use the `SUT_WINDOWS_TITLE` as the `"SUT Connector"`. When using `SUT_WINDOWS_TITLE`, the application should already be running (started manually) before starting TESTAR. Then TESTAR will search for an application that has a main window with the indicated title.

The third option is to connect through the `SUT_PROCESS_NAME`. That might be needed if the `COMMAND_LINE` is not working and the window does not have a static title to connect. When using `SUT_PROCESS_NAME`, the application should already be running (started manually) before starting TESTAR. When using `SUT_PROCESS_NAME` as the `"SUT Connector"`, TESTAR will search for a process with a process name matching the executable

of the SUT.

To test Java applications it is recommended to use `COMMAND_LINE` instead of `SUT_PROCESS_NAME`. This is because the process name of a Java application is `java.exe` and hence cannot be matched to the executable.

`SUT_WINDOWS_TITLE` and `SUT_PROCESS_NAME` allow you to start testing the SUT from a specific state, instead of the starting state of the SUT. In addition to manually starting the SUT and then starting TESTAR, it is possible to use a test script to automatically start the SUT and execute it into a specific state before (automatically) starting TESTAR.

☞ **HANDSON**: Use `SUT_WINDOWS_TITLE` and try to connect to the calculator.

☞ **HANDSON**: Use `SUT_PROCESS_NAME` and try to connect to Windows Notepad.

5.16 Execute TESTAR from the command line

TESTAR allow its execution and settings configuration from the command line, this is needed when we want to add TESTAR to a Continuous Integration pipeline. By default is executed with the selected protocol (.sse file) and the test.settings values of that protocol.

From the command line it is also possible to select the desired protocol to execute TESTAR and change the values of the test.settings.

The protocol to be executed can be selected using the "sse" parameter next to the name of the desired protocol. For example:

```
testar sse=desktop_generic
```

Other settings are input using the pairs "parameterX=valueX" separated by space. For example:

```
testar ShowVisualSettingsDialogOnStartup=false Mode=Generate
```

☞ **HANDSON**: Use the command line to execute TESTAR with the desktop generic protocol, using the SPY mode and without starting the TESTAR GUI (ShowVisualSettingsDialogOnStartup)

Some of the most interesting parameters that can help to integrate TESTAR as a Continuous Integration tool are:

`sse`: to change the protocol

`ShowVisualSettingsDialogOnStartup`: To run TESTAR without the GUI

`Mode`: TESTAR execution Mode (Spy, Generate, Record, Replay, View)

`SUTConnector` and `SUTConnectorValue`: The way to link with the desired application to be tested

`Sequences` and `SequenceLength`: The number of iterations and actions that TESTAR will execute

Certain characters such the slashes or the quotation marks must be entered in a double way to respect the treaty of special characters. For example:

```
SUTConnectorValue=" ""C:\\Program Files\\VideoLAN\\VLC\\vlc.exe"" "
```

☞ **HANDSON**: Use the command line to Generate 2 sequences of 10 actions, we are using TESTAR as a Continuous Integration tool, so we have to disable the initial GUI. Optionally you can choose any protocol and SUT that you want to test.

5.17 Play with the Tool and some other applications

The calculator is obviously very simple, we wrote it for the sake of this practical exercise. However, TESTAR can test more complex SUTs.

☞ **HANDSON**: You can change the path to another SUT if you like to see what happens when you test something like:

- Microsoft Paint
- Notepad
- VLC media player

Be careful, though, the tool might perform dangerous and unexpected actions!

5.18 Testing web applications

OneDrive is a file hosting service and synchronization service operated by Microsoft. We have already configured a test.settings file for TESTAR for OneDrive in the web_generic protocol.

Starting up TESTAR with the web_generic protocol, will use Internet Explorer to start up `https://onedrive.live.com/about/gb-en/signin/`. We get a screen that asks us to login. Evidently, letting TESTAR guess values randomly will take us a long time to enter. We would like TESTAR to enter when it starts up the system. For this we can edit the protocol that TESTAR uses to implement the test-cycle from Figure 6. In the next section this is explained.

5.19 Editing the protocol

TESTAR offers a more detailed API in the form of a Java protocol. In the "General Settings"-Tab of the TESTAR dialog you can click the "Edit Protocol" button to see and edit the default source code that the tool uses to perform its tests. You can of course use your own Java IDE to edit the source code. The source code allows you to write much more fine-grained oracles and definitions for drag and drop actions etc.

☞ **HANDSON**: Start TESTAR and, in the "General Settings"-Tab, select the web_generic protocol. Then click on "Edit Protocol". Look at the code and try to match each of the methods in the protocol with a square in the TESTAR test-cycle from Figure 6.

5.19.1 Editing the protocol to logon

Imagine we want TESTAR to automatically login to OneDrive when we start up the system (see Figure 15). For this we need to edit the web_generic protocol. To be more specific, we need to edit the `beginSequence()` method to login every time TESTAR begins to generate a sequence of our SUT (Generate and Record mode).

When editing a web protocol sometimes, we will need to to give some time to the website to render completely before you execute some Action. This can be done with:

`Util.pause(`*number*`)`

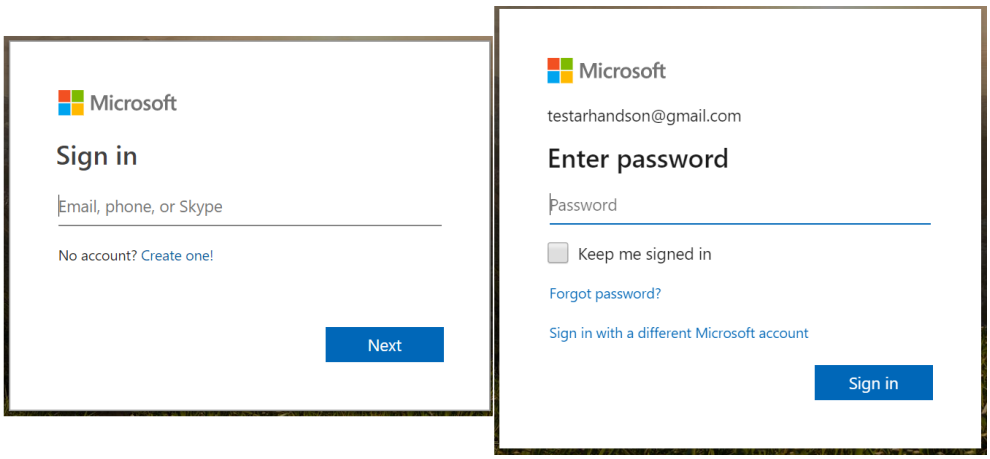

Figure 15: One Drive login web page

When we want to edit the protocol to enter some text in a text field we can do this by simulating the keyboard `presses` and `releases`. It is also important to check that when we want to enter a text into a text field, the writing of special characters into text fields depends on the user Keyboard language. The following is an example of how to write the `@` character from an ENG Keyboard.

```
import org.fruit.alayer.devices.AWTKeyboard;

Keyboard kb = AWTKeyboard.build();

//Based on ENG Keyboard, Shift + 2 typing @ character
kb.press(KBKeys.VK_SHIFT);
kb.press(KBKeys.VK_2);
kb.release(KBKeys.VK_2);
kb.release(KBKeys.VK_SHIFT);
```

☞ **HANDSON**: Let us add a piece of code so we can do the login. The TESTAR has a user account with:

username: `testarhandson@gmail.com`

password: `OneDrivetestar` (The first character is zero)

Testers have two options to prepare TESTAR for login into the SUT.

1. The first option will use the TESTAR State to read the current SUT information, search the desired Widgets and perform Actions on these Widgets, such as click on the password field and type it. NOTE: it is necessary to first update the `State` of the application (using

`getState`), to obtain the new existing Widgets. For this we can for example use the following piece of code:

```
protected void beginSequence(SUT system, State state){

  super.beginSequence(system, state);

  //Update the State
  state = getState(system);

  //Loop to search the desired Widgets
  for(Widget w :state) {

    //Check Widget with desired conditions
    if(w.get(Tags.Title,"").contains("Username")) {

        //Create a TESTAR action compiler
        StdActionCompiler ac = new AnnotatingActionCompiler();

        //Create a type action with the desired text
        Action a = ac.clickTypeInto(w, "username", true);

        //Execute the action on the State of our system
        executeAction(system, state, a);
    }
}

//Wait a bit
Util.pause(5);

}
```

2. The second option, the simplest, is to specify a sequence of Keyboard Actions to navigate and input data in the required fields. To enable keyboard navigation we will use a `CompoundAction` builder, that contains a series of actions that you want to execute when the system starts up. Note that, in the following example, when the username is being typed, it is assumed that keyboard focus is on the user field, but this may not be the case when you start up. So you might first need to TAB your way to the right widget.

```
new CompoundAction.Builder()
    // assume keyboard focus is on the user field
    .add(new Type("my␣user"),0.1)

    // assume next focusable field is pass
    .add(new KeyDown(KBKeys.VK_TAB),0.5)
    .add(new Type("my␣user␣pass"),0.1)
```

```
        // assume login is performed by ENTER
        .add(new KeyDown(KBKeys.VK_ENTER),0.5).build()

        .run(sut,null,0.1);
    return sut;
```

5.19.2 Editing the protocol to add an oracle

If we want to add oracles other than those that can be represented with the regular expressions, we can edit the `get_Verdict()` method.

Let us look at a small example, where we want to check in every state that the widgets that contains some text (i.e. its NativeRole is "UIAText") can fit a font size that is at least the `MINIMUM_FONT_SIZE`. This can be done by writing the following piece of code returning a `Verdict`.

```
Verdict getSmallTextVerdict(State state,
                            Widget w,
                            Role role,
                            Shape shape){
  final int MINIMUM_FONT_SIZE = 8; // px
  if (role != null
      && role.equals(NativeLinker.getNativeRole("UIAText"))
      && shape.height() < MINIMUM_FONT_SIZE
     )
    return new Verdict(SEVERITY_WARNING,
                       "Not␣all␣texts␣have␣a␣size␣greater␣than␣"
                       + MINIMUM_FONT_SIZE + "px");
  else return Verdict.OK;
}
```

Then we can call this method in a `get_Verdict` while loop for every state.

```
Role role;
String title;
Shape shape;

// apply the oracles to all widgets
for(Widget w : state){
  role = w.get(Tags.Role, null);
  title = w.get(Title, "");
  shape = w.get(Tags.Shape, null);

  // check for too small texts to be legible
  verdict = verdict.join(getSmallTextVerdict(state, w, role, shape));
}
```

```
return verdict;
```

HANDSON: Try to add a piece of code that defines an oracle that checks whether every image on the screen (i.e. its NativeRole is "UIAImage") has an additional textual description according to the WAI guidelines for accesible webpages.

5.20 Testing Java Swing applications

HANDSON: Try to test the JEdit application with the generic protocol (Run `C:\Program Files\jEdit\jedit.exe` and configure TESTAR to connect with `SUT_Windows_Title` option). What is happening? This is a Java Swing application and the accessibility API does not work for those. It does not give TESTAR the information it needs to get the current state and all the widget information.

`Java AWT` is a heavy-weight technology because its components are dependent on the underlying Operating System. These components are represented in the native OS and that's why the Windows API can detect them. However, by default, `Java Swing` components only exist at the `Java Virtual Machine` level and we cannot find them at the native OS level.

To test Java Swing applications, it is necessary to make some changes in the Windows operating system configuration to enable the Java Access Bridge technology. This can be done from the Windows control panel as follows:

Control Panel → Ease of Access → Ease of Access Center → Make the computer easier to see → Other installed programs → Enable Java Access Bridge

Once the Java Access Bridge technology is enabled in the operating system, the `"AccessBridgeEnabled"` variable must be set to `"true"` in the `test.settings` file.

Note: this only works when you are using Oracle JDK1.8. For openJDK, this is not available yet. To see which Java you have you can use `java -version`.

```
#####################################################################
# Java Swing applications & Access Bridge Enabled
```

```
#
# Activate the Java Access Bridge in your Windows System:
# (Control Panel / Ease of Access / Ease of Access Center /
# Make the computer easier to see)
#
# Enable the variable Access Bridge Enabled in TESTAR as true
#################################################################

AccessBridgeEnabled = true
```

☞ **HANDSON**: Change the configuration of the Windows Operating System by enabling the Java Accesibility Bridge. Change the `test.settings` file of `Desktop generic` to enable the Java Access Bridge. Now you can test for example JEdit and implement the needed action filters.

We could edit the test settings for `AccessBridgeEnabled` from false to true for Swing applications and back again for non-Swing applications. However, we can also make a new protocol only for Swing applications. This will make switching between different type of applications a lot easier. Adding a new protocol is done as follows.

In the settings folder (`\testar\bin\settings`), we create a new folder named, for example, `desktop_swing`. In this directory we add the files `Protocol_desktop_swing.java` and `test.settings`. We can edit those files to prepare a specific configuration for the Java Swing applications, and avoid the constant modification of the rest of the protocols.

☞ **HANDSON**: Create a new protocol folder to separate configuration of generic applications and Java Swing applications. Modify the new protocol files to prepare a configuration for Java Swing. Select your new protocol and check that it works correctly.

For some widgets of swing applications, the accessibility API infers incorrect values for the *Role* property, e.g. it can infer that the *Role* is "Text" while it is actually a "Clickable" item. This happens for example for widgets that inherit from UIAList, UIAComboBox and UIATree. To understand better what is going on here, we will guide you through the following modifications using JEdit as the example SUT.

☞ **HANDSON**: Start TESTAR with JEdit in SPY mode, and enable the visualization of green dots (remember this was done with Shift + ⇧). Now navigate to the Menu Utilities and click on Global Options. Inspect the widgets that belong to the menu on the left. You can see that the

widgets belonging to this menu are correctly detected by TESTAR, but their accessibility properties indicate that these are plain text (UIAText) while it should be a clickable widget.

By editing the protocol, we can modify TESTAR to customize the action derivation of the application we want to test. Therefore we will add the following modifications to allow actions on these widgets. In the derive actions method, when iterating through the enabled widgets, we will add actions to these problematic widgets.

```
//Force actions on some widgets with a wrong accessibility
//This is optional, you can comment this out if your Swing
//applications doesn't need it.
    if(w.get(Tags.Role).toString().contains("Tree")
    ||
    (w.get(Tags.Role).toString().contains("ComboBox")
    ||
    (w.get(Tags.Role).toString().contains("List")
    ) {
        widgetTree(w, actions);
    }
```

We will add the widgetTree method, prepared to add the action `leftClickAt(widget)`, after the method `deriveActions`:

```
//Force actions on widgets with a wrong accessibility
public void widgetTree(Widget w, Set<Action> actions) {
   StdActionCompiler ac = new AnnotatingActionCompiler();
   actions.add(ac.leftClickAt(w));
   w.set(Tags.ActionSet, actions);
   for(int i = 0; i<w.childCount(); i++) {
      widgetTree(w.child(i), actions);
   }
}
```

☞ **HANDSON**: Make the changes in the swing protocol. Compile the protocol verifying that there are no faults. Open JEdit and verify that these widgets are now enabled to perform some click action Shift + 1

5.21 TESTAR's generated statemodel

While testing, TESTAR generates a state model which it stores in an OrientDB graph database. After the tests, this database can be queried.

5.21.1 Using the graph database

TESTAR has the capability to export the discovered states, widgets and executed actions to an OrientDb graph database. The database can be used interactively.

HANDSON: Run a test sequence on a SUT with GraphDB enabled by executing the following 3 steps:

1. Set parameters in the "GraphDB"-Tab of the TESTAR Dialog or test.settings file.

   ```
   ######################################################################
   # GraphDB
   #
   # parameters for storing the graph representation in OrientDB
   ######################################################################
   GraphDBEnabled = true
   GraphDBUrl = plocal:/local/orientdb/databases/<dbname>
   GraphDBUser = admin
   GraphDBPassword = admin
   ```

   <dbname> is the user-defined filename of the database.

2. Review the test protocol (through the TESTAR Dialog choosing 'Edit Protocol' button). Make sure that the protocol when deriving actions, stores the widgets in the graph database.

   The *desktop_generic* protocol already takes care of this requirement. Other protocols may need to be revised. This can be done by calling the *storeWidget* method in the *deriveActions* method of the protocol. The code below is for illustration, it is not intended as the complete implementation of the *deriveActions* method.

   ```
   protected Set<Action> deriveActions(SUT system, State state)
                         throws ActionBuildException{
      //... skeleton iterate through all widgets
      for(Widget w : state){
              storeWidget(state.get(Tags.ConcreteID), w);
      //..
   }
   ```

   A reference implementation can also be copied from the desktop_generic protocol.

3. Click on the Generate button to start the test

After TESTAR has completed its test sequence the Graph database can be used to analyse and query the results.

☞ **HANDSON**: Analyse the test results with OrientDB by doing the following steps:

1. Start the OrientDB Server.
   This can be done by clicking on the *OrientDB Server* Icon on the desktop. This will launch the server at port 2480.
2. Connect to the graph database.
   Open a browser with url `http://localhost:2480`, choose the database that TESTAR populated and supply the user "admin" with password "admin" .
3. Explore the graph database.
   From the top menu select 'Graph' and at the Graph Editor enter on line 1: `'select * from V'`. Subsequently, press the play button, this will render the graph. Orient studio is limited by a display of 20 nodes by default. To change this to something bigger, like 1000, click the wrench icon (right part of the screen) and alter the value for 'Limit'.
4. Stop the Server.
   In the server-command window, enter Ctrl-C and stop the batch job

Notes:

- The OrientDb graph editor is suitable for structures up to a few hundred nodes and edges. Larger ones will end up as a hairball graphs.
- Filtering can be achieved by adding a Where-clause to the query like: s*elect from V where (not (@class in ['AbsState', 'AbstractAction', 'AbsRoleTitle', 'AbsRoleTitlePath', 'AbsRole']))*
- Annotations can be altered by clicking a node (or edge) and using the settings panel in the left navigation bar.
- TESTAR can raise an Exception "<Cannot open local storage...".
  This occurs when the database was created on a previous run and server is still running. The solution for this is: Either stop the server and rerun the test sequence or leave the server running and just rename the existing database to a new one.

☞ **HANDSON**: Try to find the Starting node by executing the proper query.

6 WebDriver

The webdriver version of TESTAR is developed to better handle web applications. For various reasons, the desktop version doesn't recognize the non-native widgets available in websites. TESTAR webdriver tries to remedy this by looking at the DOM and extracting the information via a (JS) web extension, added to the browser. The Selenium webdriver acts as a bridge between the browser and the webdriver module of TESTAR.

Apart from the improved detection of widgets on web applications, using the wedriver module allows the user to use TESTAR 'natively'. As all interactions are on the browsers viewport, it is not necessary to use a VM. The webdriver version of TESTAR is tested on 3 mayor platforms: Windows 10, OS X and Linux. It has been tested with Chrome/Chromium, Firefox and Edge (Windows). That said, using Chrome seems to be the best option.

6.1 Installing the Selenium Webdriver

For TESTAR to be able to connect to a browser, it needs the Selenium Webdriver. It acts as a bridge between TESTAR and the browser of your system. It is a W3C standard, and is supported by the 3 biggest browsers : Chrome/Chromium, Edge and Firefox. Unless there is a specific requirement for another browser, the driver for Chrome is recommended as it has the best performance. The drivers can be found here:

- Chrome/Chromium
  `http://chromedriver.chromium.org/downloads`
- Firefox (Geckodriver)
  `https://github.com/mozilla/geckodriver/releases`
- Edge (Microsoft Webdriver)
  `https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/#downloads`

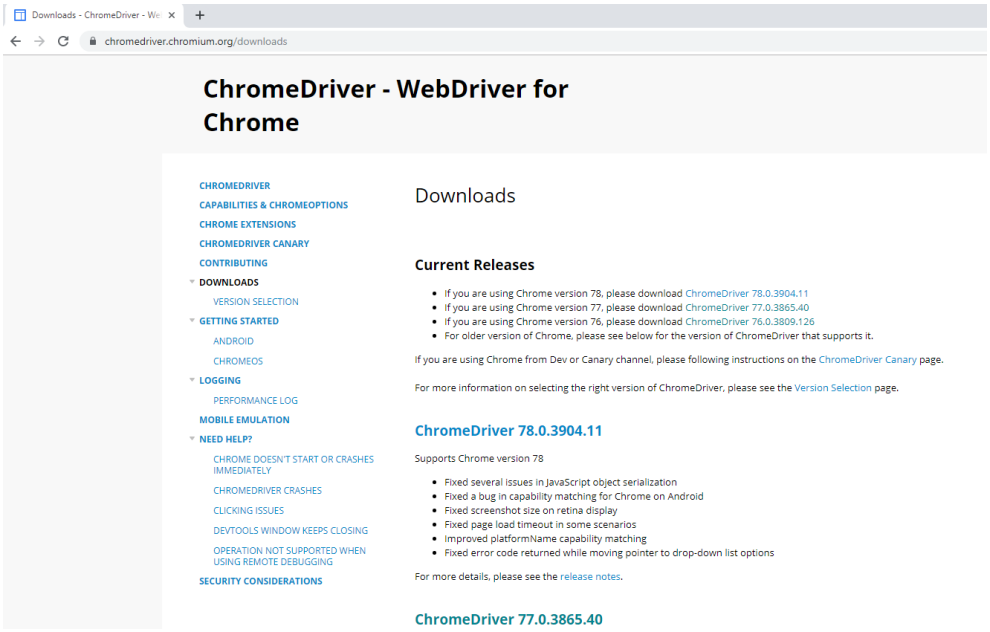

Figure 16: Download Selenium ChromeDriver

In the indicated URLs the different versions of the webdrivers are updated. When we are going to select a version to download, it is important to consider the current version of our browser (`chrome://settings/help`).

`NOTE:` If the versions are incompatible and you need to download a new chromedriver, TESTAR will display a warning message.

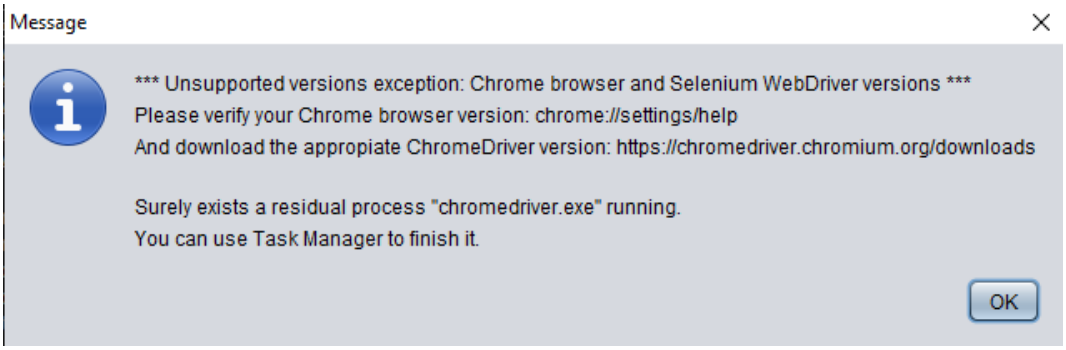

Figure 17: Warning message: ChromeDriver version error

For all variants, it is important to download the version that belongs

to the browser installed on your system. The location where you install the driver is important, as it needed in the TESTAR configuration. Sensible locations would be :

- Windows : `C:\Windows\chromedriver.exe`
- OS X : `/usr/local/bin/chromedriver`
- Linux : `/usr/lib/chromium-browser/chromedriver`

☞ **HANDSON**: Install the Selenium Webdriver for Chrome on your system

6.2 Start TESTAR

In the bin directory of the extracted directory, you can find the startup scripts for TESTAR. For Windows use `testar.bat`, for OS X / Linux use `testar`. These can be opened in 2 ways :

- Navigate to the `bin` directory in the filemanager of your system
- Doubleclick the script to start the GUI

- Navigate to the `bin` directory in a command prompt or terminal
- Run the script for your system by typing : `testar.bat` or `./testar`

☞ **HANDSON**: Start the TESTAR GUI

NOTE: If your `JAVA_HOME` path does not point correctly to the installation path for the Java Development Kit you will see a

```
java.lang.RuntimeException: JDK required (running inside of JRE)
```

To set your `JAVA_HOME` look for example on stackoverflow[5] for the instructions for your type of machine.

NOTE: sometimes, if the configuration is not complete, something like Figure 14 from Section 13 is shown first. Then, at this point, you should select the *desktop_generic* and upon pressing OK, the GUI from TESTAR like in Figure 3 starts up.

[5] `https://stackoverflow.com/questions/2619584/how-to-set-java-home-on-windows-7`

6.3 Configure webdriver path

When TESTAR has started correctly, the GUI will resemble the one in Figure 3. The main differences will be that the SUT path (the text in the textarea) will consist of 3 parts (see Figure 18):

- Selenium webdriver path: indicates the PATH towards the location of the Selenium Webdriver. If this differs from the path you used in section 6.1, change this here.
- Screen dimensions (optional), indicates the size to use for the browser. Set this to the size of your screen, or smaller to allow a task bar at the bottom of your screen. Don't change the '+0+0' part, as this tells TESTAR to place the browser in the upper left corner.
- URL of the webapplication you want to test.

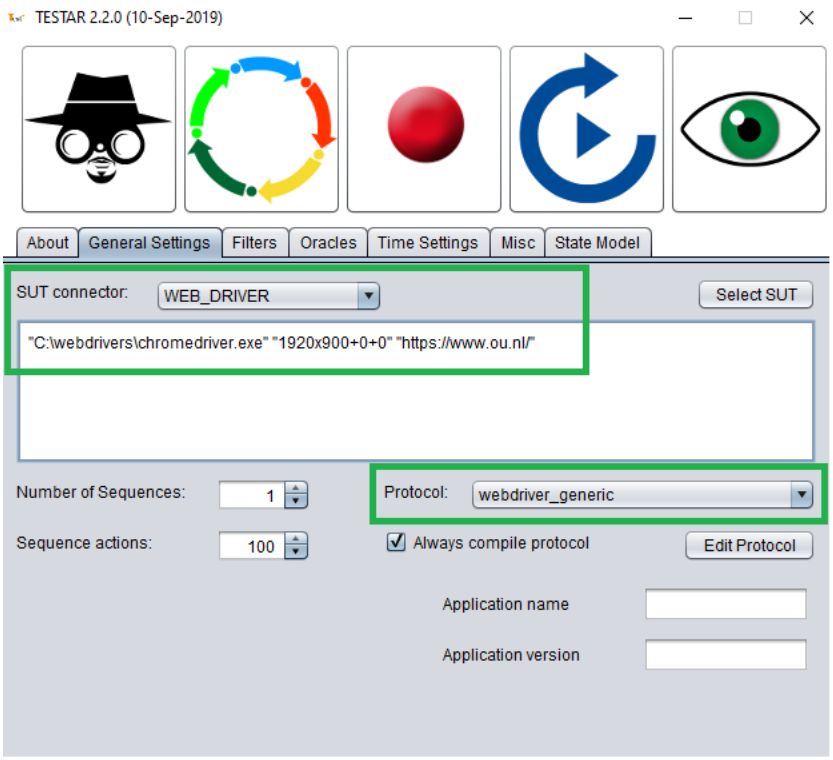

Figure 18: Use TESTAR WebDriver protocol

HANDSON: Configure the Selenium Webdriver path.

Once the path to the Webdriver is set (and the webdriver_generic protocol is selected), you can start TESTAR for example in SPY mode. TESTAR should disappear, and Chrome should open the URL of the website that we indicated we want to test (e.g. in the example `https://www.ou.nl/`).

The loaded page should look like Figure 19, with the notification `Chrome is being controlled by automated test software`, denoting that the browser is started by Webdriver. The TESTAR logo next to the address bar shows that a browser extension [6] (also known as add-on) is loaded. This extension comes as part of the TESTAR installation, and provides a communication channel between the browser and the Java code. This channel conveys data (e.g. page state) and actions (e.g. a `back` action).

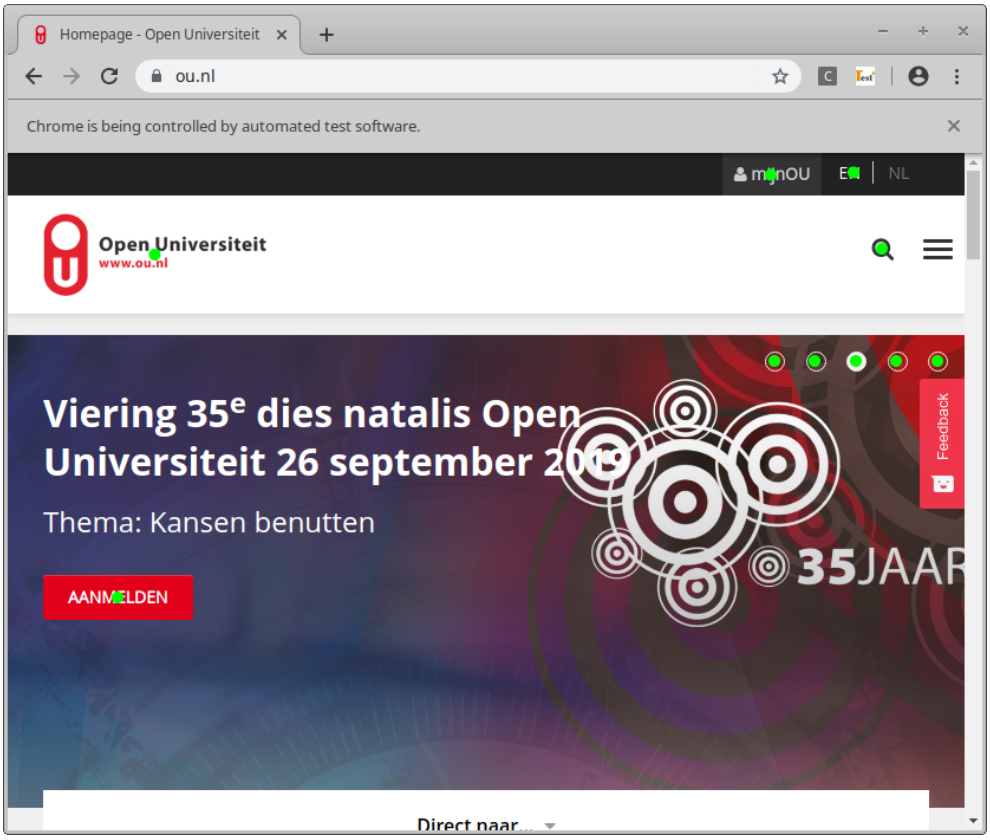

Figure 19: The TESTAR logo indicates the installed extension

☞ **HANDSON**: Verify that the extension and the OU page are loaded.

[6] More info `https://en.wikipedia.org/wiki/Browser_extension`

6.4 Discover webdriver features

Click the `Edit protocol` button to access the source code of the webdriver protocol. Into this Java protocol, the user can customize some webdriver features:

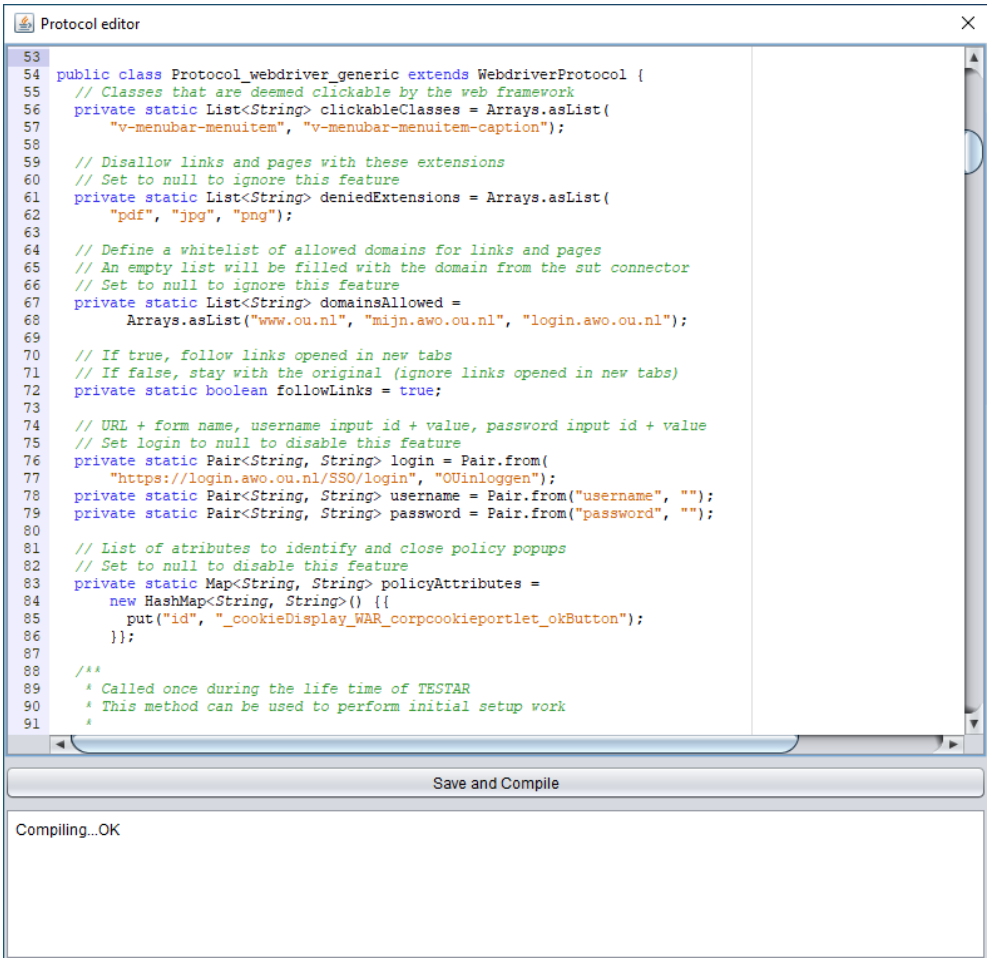

Figure 20: Webdriver protocol features

6.4.1 The whitelisting of domains

One of the features of using TESTAR with the webdriver, is the whitelisting of domains. This feature is implemented with the intention of keeping our tests in the same web domain, since many web applications have links that point to external domains.

Its purpose can be seen if you scroll down to the footer of the webpage, which mostly contains links to pages within the `https://ou.nl` domain. These are all regarded clickable by TESTAR, and thus decorated with the familiar green dot. But the link to `Ict status en onderhoud` (`Ict status and service`) stands out. Its URL `http://oustatus.nl/` is not in the whitelisted domains, and will therefore not be tested. Adding the domain to the class variable `domainsAllowed` allows the link to be visited. Alternatively, setting `domainsAllowed` to `null` overwrites this feature and allow all domains to be tested.

☞ **HANDSON**: Add the status domain to the `domainsAllowed` list in the Java protocol.

6.4.2 The blacklisting of extensions

A complementary feature is the blacklisting of extensions, for example pages with images or PDFs, which do not have any interactive elements. Testing these pages can cause TESTAR to get stuck and consume resources. This feature can be illustrated when you browse to `https://www.ou.nl/jaarverslagen`. All the links to the various annual reports are ignored by TESTAR due to the extensions of the target URLs. Removing the `pdf` entry to the `deniedExtensions` parameter list will allow TESTAR to regard them as clickable.

☞ **HANDSON**: Remove `pdf` from the `deniedExtensions` list

6.4.3 Add clickable elements

Another powerful feature is realised by the class variable `clickableClasses`. Sometimes pages contain non-native elements which are nevertheless clickable. As TESTAR can only detect native elements (hyperlink, buttons etc.), another mechanism is needed to designate widgets as clickable. Looking at `https://www.ou.nl`, we see the menu items and tiles recognized as clickable due to them being hyperlinks. But the `Feedback` widget on the right doesn't have green dot, as it is a `<div>` element, with Javascript providing the interactivity. To find the CSS-class, open the `DevTools` (`F12`) and select `Inspect` the `Feedback` element via the right-click menu. Make sure to get the inner `<div>` containing the text. Hint: the CSS-class ends with '_label_text'.

☞ **HANDSON**: Add the `Feedback` CSS-class to `clickableClasses`.

After adding the class, the `Feedback` widget is nicely decorated with a green dot. Unfortunately, this is not the case for the emojis if the Feedback-panel is unfolded. As in the previous paragraph, these are not native elements, but in this case `<span>` tags. To make them clickable we need to add their common CSS-class to the `clickableClasses`.

HANDSON: Add the `Emoji` CSS-class to the `clickableClasses` list.

Here you can see an example of the `clickableClasses` customization made in the `webdriver_gwt` protocol.

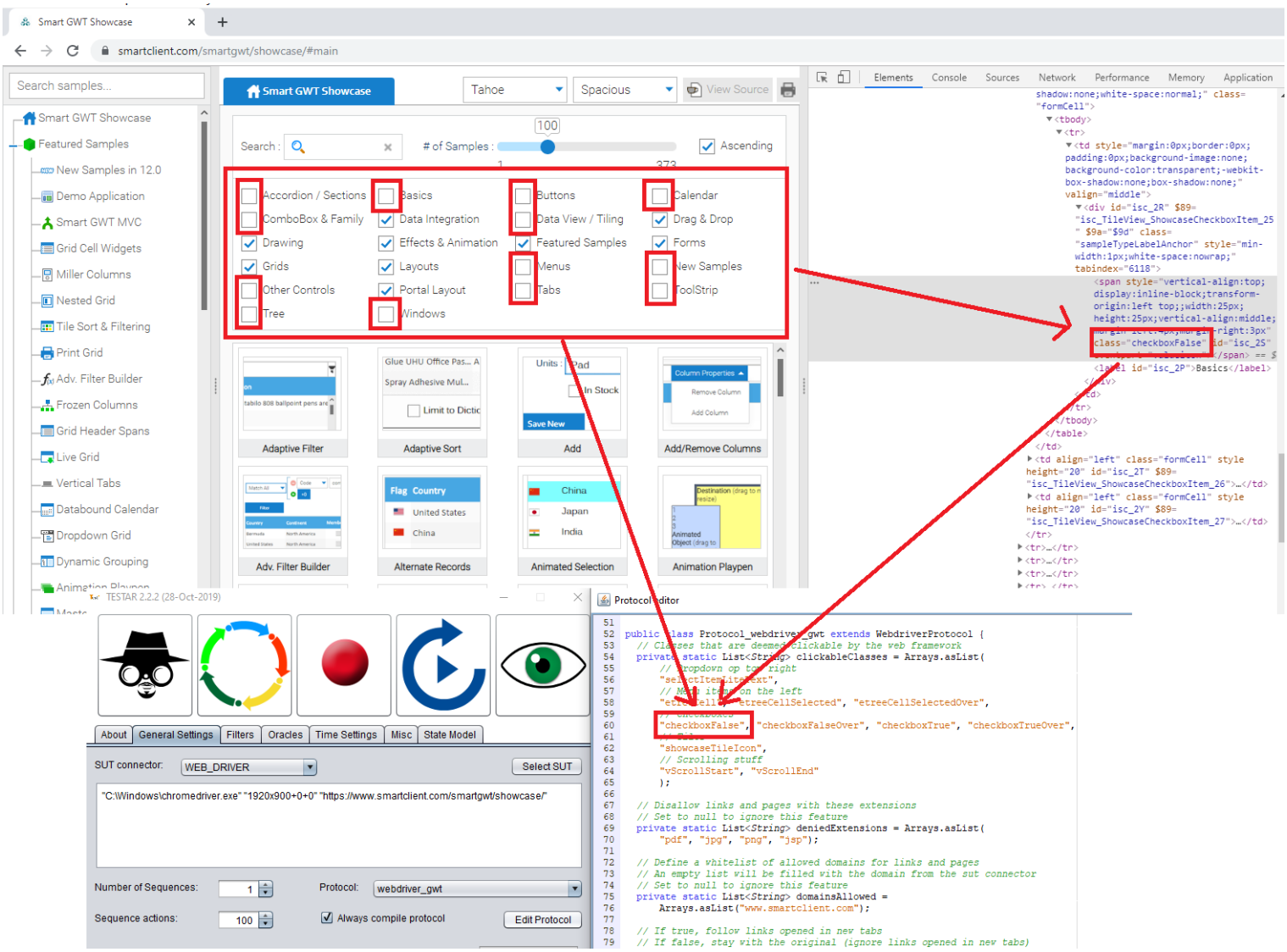

Figure 21: Example of Css Classes customization from GWT protocol

6.4.4 Blocking overlays and password protection

To demonstrate the last 2 features, go to the `C:\\testar\\bin\\suts` directory and locate the file

`cookie\_login\_example.html`

Browsers can open files from the filesystem by using the `file://` protocol instead of `https://`. Thus the new URL should be `file:///C:\testar\bin\suts\cookie_login_example.html`, ensure this is the correct location.

☞ **HANDSON**: Make TESTAR open the `cookie_login_example.html` page by replacing `https://www.ou.nl/` with the new URL

When you run TESTAR in SPY mode, you will see 2 familiar obstacles for modern web browsing: blocking overlays (often used for cookie laws) and password protection. Although necessary, they block automated testing and therefore require additional TESTAR features to handle them.

For the first obstacle the `policyAttributes` parameter is available, taking sets of tag attribute names and values. Find the name and value needed to close the overlay in the source code of the page, and add them to the `policyAttributes` parameter.

☞ **HANDSON**: Find the attribute name and value of the close button and add to the `policyAttributes` parameter

After the overlays requesting permission for cookies, asking for login credentials is the next obstacle for automated testing. The webdriver protocols provide three `Pair` objects to deal with this. The first `login` pair requires a URL (incl. protocol) and a form name, the latter is submitted once the credentials are provided. These can be defined in the next pairs, `username` and `password`, a combination of the element id and its value.

☞ **HANDSON**: Find the needed credentials in the page source and fill in the 3 parameter `login`, `username` and `password`

A Keyboard shortcuts

Several keyboard shortcuts are available for the different TESTAR working modes (i.e. SPY, GENERATE, RECORD and REPLAY):

Keyboard shortcut	**Effect**	**Working modes**			
		Spy	Generate	Record	Replay
Shift + Arrow Up	Show widgets info	✓	✓	✓	
Shift + Arrow Down	Close the mode and go back to TESTAR dialog	✓	✓	✓	✓
Shift + Arrow Right	Switch the working mode		✓	✓	
Shift + Space	Toggle slow motion test		✓		✓
CAPS_LOCK/TAB + (Shift) Ctrl	UI widgets' actions filtering	✓			

Table 1: Keyboard shortcuts

B Directories

./settings	Tests set ups
./output	Reports of the different TESTAR runs
./output/`timestamp_SUTname`	Reports: logs, screenshots, graphs, metrics, serialized tests
././sequences	All the serialized test sequences
././sequences_V	Classified test sequences by verdict V. Where V can be from {ok, suspiciostitle, unexpectedclose}
././HTML reports	Visual html report
././srcshots	Screenshots of tests UI states and executed UI actions
././logs	Tests logging data
./output/graphs	Tests graphing for visual analysis
./output/metrics	Tests performance indicators
./output/temp	Temporary files such as the last recorded test sequence

C Test settings

- **ActionDuration** = a non-negative decimal. Sets the speed, in seconds, at which an UI action is performed. For example, typing a text will introduce delays between each key stroke.
- **AlgorithmFormsFilling** = true or false. Enables or disables a specific UI action selection algorithm that will try to populate data in UI forms.
- **ClickFilter** = regular expression. Prevents UI actions to be performed on UI elements whose *TITLE* matches the regular expression. The rationale behind this is that certain UI actions might be dangerous or undesirable without human supervision (e.g. printing documents, files operations). For example: .*[cC]lose.*|.*[eE]xit.*|.*[pP]rint.*).
- **CopyFromTo** = (source_file_path;target_file_path)*. A list of (>=0) pairs of source and target files to copy before a test starts (click the text-area and a file dialog will pop up). Sometimes, it can be useful to restore certain configuration files to their default prior to SUT execution, so that the SUT starts from a desired state.
- **Delete** = (file_path)*. A list of (>=0) files to delete before a test starts (click the text-area and a file dialog will pop up). Certain SUTs may generate configuration files, temporary files and/or files that save the SUT' state. Thus, you can restore your SUT environment to a desired state removing files generated from previous executions.
- **DrawWidgetInfo** = true or false. Sets whether to display detailed overlay information, inside the *Spy mode* over the selected widget in UI of the SUT.
- **DrawWidgetTree** = true or false. Sets whether to display a graphical representation of the widget-tree, inside the *Spy mode* for the selected widget in the SUT' UI.
- **DrawWidgetUnderCursor** = true or false. Sets whether to display brief overlay information, inside the *Spy mode* over the selected widget in the UI of the SUT.
- **ExplorationSampleInterval** = a positive number. Sets the metrics sampling interval by the number of executed UI actions during a test.
- **ForceForeground** = true or false. Sets whether to keep the SUT' UI window active in the screen (e.g. when its minimised or when a process is started and its UI is in front, etc.).

- **ForceToSequenceLength** = true or false. Setting the value to true, if a test fails (e.g. the SUT crashes), TESTAR continues the test sequence until it reaches the specified test sequence length (check *SequenceLength* property). Otherwise (false value), the test will finish in the presence of a fail.

- **MaxTime** = a positive number. Sets a time window, in seconds, after which the test is finished (e.g. stop after an hour, a day or a week).

- **Mode** = Spy, Generate or GenerateDebug (check *ShowVisualSettingsDialogOnStartup* property). Runs the tool into the Spy, Generate or GenerateDebug mode.

- **NonReactingUIThreshold** = a positive number. Sets a test window (number of UI actions) for which a non-reacting UI will force to perform UI actions that could potentially make the UI to react (e.g. an ESC key stroke to close a popup dialog box).

- **LogLevel** = 0, 1 or 2. Sets the logging level to critical messages (0), information messages (1) or debug messages (2).

- **OnlySaveFaultySequences** = true or false. Sets whether to save non-fail test sequences.

- **ProcessesToKillDuringTest** = regular expression. Any process name that matches the regular expression and is started during a test will be automatically killed. The rationale behind this is that some UI actions could start undesirable processes (e.g. an email client). For example: (e.g. .*[oO]utlook.*|firefox.exe).

- **ProtocolClass** = settings_folder/Test_protocol_class_name. Links to the test protocol class under the folder denoted by the *MyClassPath* property.

- **ReplayRetryTime** = a positive number. Inside the replay mode, establishes the time window in seconds for trying to replay a UI action of a replayed test sequence.

- **SequenceLength** = a positive number. Sets each test sequence (check *Sequences* property) length as the number of UI actions to perform[7]. Check the *StopGenerationOnFault*, *ForceToSequenceLength*, *MaxTime*, *SuspiciousTitles*, *TimeToFreeze* and *ProtocolClass* properties for specific behaviour.

- **Sequences** = a positive number. Number of times to repeat a test.

[7]Note: higher values will consume more hardware resources, specialy if graphing was activated.

- **ShowVisualSettingsDialogOnStartup** = true or false. Sets whether to display the tool UI. If false is used, then the tool will run in the mode of the *Mode* property.

- **StartupTime** = a positive number. Sets how many seconds to wait for the SUT to be ready for testing (its UI being accesible by TESTAR). If the SUT did not start on time the test will not run. Otherwise, test will start as soon as the UI is accesible. Take into account that the first time the SUT is run on your environment will usually take more time than next executions (e.g. due to memory catching).

- **StopGenerationOnFault** = true or false. Sets whether to finish a test in the presence of a fail (e.g. a SUT crash). Setting it to false does not necessarily mean that the test will continue, but the test will try to continue as far as the SUT accepts additional UI actions and the test set up does not finish the test by other means (e.g. *MaxTime*, *SuspiciousTitles*, *TimeToFreeze* or *ProtocolClass* properties).

- **SUTConnector** = COMMAND_LINE, SUT_WINDOW_TITLE or SUT_PROCESS_NAME. Sets the approach used to connect with your SUT:

  - COMMAND_LINE: *SUTConnectorValue* property must be a command line that starts the SUT. It should work from a Command Prompt terminal window (e.g. *java -jar SUTs/calc.jar*). For web applications follow the next format: *web_browser_path SUT_URL*.
  - SUT_WINDOW_TITLE: *SUTConnectorValue* property must be the non-empty *title* displayed in the SUT's main window. The SUT must be manually started and closed.
  - SUT_PROCESS_NAME: *SUTConnectorValue* property must be the process name of the SUT. The SUT must be manually started and closed.

- **SUTConnectorValue** = check *SUTConnector* property.

- **SuspiciousTitles** = a regular expression. Checks the UI for any suspicious title that could denote problems in the SUT. TESTAR checks whether there exists a widget' *TITLE* in the UI that matches the regular expression. If a match was found the test will continue but you will find the issues found in the reports. For example, a critical message like "A NullPointerException Exception has been thrown" can be represented by the regular expression ".*NullPointerException.*".

- **TimeToFreeze** = a positive number. Sets the time window, in seconds, for which to wait for a not responding SUT. After that, the test

will finish with a fail. The rationale behind this is that the SUT could hang, be performing heavy computations or be waiting for slow operations (e.g. bad internet connection). The value of the property is thus a threshold after which the SUT is interpreted to have hung.

- **TimeToWaitAfterAction** = a non-negative decimal. Sets the delay, in seconds, between UI actions during a test. It directly affects the reproducibility of tests and tests performance. Setting it to a low value will speed up the tests, but the SUT could not have finished processing an action before the next action is executed by TESTAR. In the latter case the test could not be reproducible, but it could reveal potential faults (stress testing).

- **UseRecordedActionDurationAndWaitTimeDuringReplay** = true or false. Inside the replay mode sets whether to use the action duration (check *ActionDuration* property) and action delay (check *TimeToWaitAfterAction* property) as specified in the recorded test sequence. If set to false, the values from the current set up are used.

- **VisualizeActions** = true or false. Sets whether to display overlay information, inside the *Spy mode*, for all the UI actions derived from the test set up.

Index

action filter
  clickfilter, 22
  regular expression, 21

clickfilter feature, 22
CompoundAction builder, 31
Continuous Integration, 27

Dialog Tab, *see* TESTAR Dialog Tabs

filter actions
  clickfilter, 22
  regular expression, 21
FITTEST, 5

general-purpose requirements, 14
GENERATE Mode, 14, 15
graph
  database, 36
GUI, 5

Mode
  GENERATE, 14, 15
  RECORD, 20
  REPLAY, 19
  SPY, 12

oracle, 20
  implicit, 20
  suspicious title, 20
orientdb, 36

protocol, 29
  edit, 29
  method
    beginSequence(), 29
    finishSequence(), 29
    get_Verdict(), 32
  Protocol_desktop_generic.java, 22
  web_generic, 29
protocol_filter.xml, 22

RECORD Mode, 20
regular expression, 21
REPLAY Mode, 19

SPY mode, 12
sse file, 22
stopping criteria, 14
SUT, 5
  Calculator (buggy), 6
  connector, 12, 26
    COMMAND_LINE, 12, 26
    SUT_PROCESS_NAME, 26
    SUT_WINDOWS_TITLE, 26
  MS Paint, 28
  Notepad, 28
  OneDrive, 29
  VLC media player, 28
Swing applications, 33

Tab, *see* TESTAR Dialog Tabs
test.settings file, 11, 22
TESTAR
  Dialog Tabs
    Filters Settings, 21, 22
    General Settings, 12, 15, 24, 26
    GraphDB, 36
    Oracles, 20, 21
  Protocol, *see* protocol
  web (www.testar.org), 5

Verdict, 32