Automated Testing at the GUI level

Hands-on do it yourself session

2018

The TESTAR team

Contents

1 What is GUI Testing? 4

2 We start with a simple buggy SUT 4

3 The Virtual Machine 4

4 Hands-on step by step 5

4.1 Manually test the SUT 5

4.2 Starting up TESTAR 7

4.3 The SPY Mode 7

4.4 The GENERATE-TEST mode 8

4.5 Design a Test Oracle 10

4.6 Adjust TESTAR's Behavior 11

4.7 The test.settings file 12

4.8 Process Listener 13

4.9 Failure BINGO! 15

4.10 Connecting with a SUT: the SUT connectors 16

4.11 Play with the Tool and some other applications 17

4.12 Testing web applications 17

4.13 Editing the protocol 17

4.13.1 Editing the protocol to logon 18

4.13.2 Editing the protocol to add an oracle 18

4.14 Testing Java Swing applications 19

4.15 Some ways to Analyse TESTAR output 22

4.15.1 Viewing executed test sequences 22

4.15.2 Offline test graphs 23

4.15.3 Using the graph database 24

A Keyboard shortcuts **27**

B Directories **28**

C Test settings **29**

D Calculator Failures BINGO! **33**

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 throughout the test. This type of testing is usually 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...

Therefore, in this assignment we will try to automate GUI testing by using a tool called *TESTAR* [1]. TESTAR is an open source tool for automated testing through GUI. It automatically interacts with the SUT by clicking and typing on the controls of the GUI. It is able to recognize abnormal SUT behaviour, such as crashing or freezing, and you can add application specific test oracles. 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 stresses GUI-based applications and can – once setup 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 handson we created a virtual machine with software that is needed already installed. To run the virtual machine you will need to run virtualbox version 5.2.10 or later. If you do not have VirtualBox 5.2.10 on your machine you can download VirtualBox 5.2.10 here: `https://download.virtualbox.org/virtualbox/5.2.10/` .

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

If you have a newer version ($>$ 5.2.10) of VirtualBox installed, you will need to update the "VBox Guest Additions" on the virtual machine 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 $\geq$ 5.2.10
- Recent Dual core CPU or better
- 8 GB Memory
- about 80 GB free diskspace

The handson OVA-image can be downloaded from the Testar website (`https://testar.org/images/`).

The OVA-image is about 25GB downloaded.

☞ **HANDSON**: Import the downloaded OVA-image in VirtualBox via "File" $\rightarrow$ "Import Applicance"

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

The Username on Windows 10 is "`testar`" and the password is "`testar`".

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

4 Hands-on step by step

4.1 Manually test the SUT

Before using TESTAR, test the SUT manually, to get an impression of potential failures. You can find the Calculator in `C:\local\suts\Calculator.jar`. You can either double click on the `Calculator.jar` file or in the command prompt type `java -jar C:\local\suts\Calculator.jar`.

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

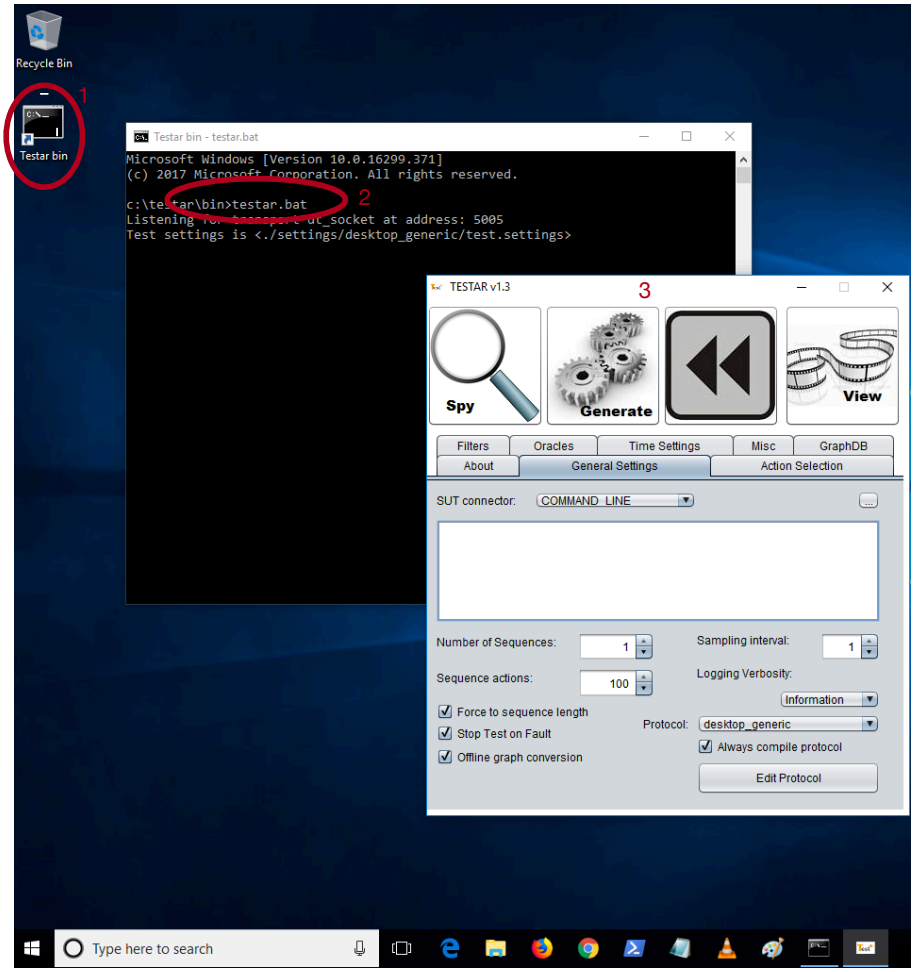

Figure 1: Starting TESTAR in the Virtual Machine

4.2 Starting up TESTAR

Now let us start up TESTAR to see what it can do. On the desktop of the virtual machine you see a shortcut to `Testar bin` (see 1 in Figure 1). Double click on that and the windows command prompt will open. Type `testar.bat` (like 2 in Figure 1) and on Enter the GUI of TESTAR starts up (like 3 in Figure 1).

Basically the GUI of TESTAR is a dialog that enables us to configure the values that are present in the test.settings file. These settings define details about the SUT that tell TESTAR what and how to test. For example, in the "General Settings"-Tab you can see a SUT connector, that is a setting that defines the way that TESTAR connects to the SUT. The default is set to connect through `COMMAND_LINE`. In the text field you can now type the following command to indicate to TESTAR that you want to connect to the Calculator as a java jar:

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

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.

4.3 The SPY Mode

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

- press Shift + 1 to 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.
- press Shift + 2 and hover over an element allows to show or hide the accessibility information of the widgets.
- press Shift + 3 and hover over an element it will show detailed information about that element. This way you can find for example the titles of the elements. To go back to less information just press Shift + 3 again.

To stop TESTAR, press Shift + ↓. You can find more shortcuts in Appendix A. It is a good idea to become familiar with this Mode and its shortcuts.

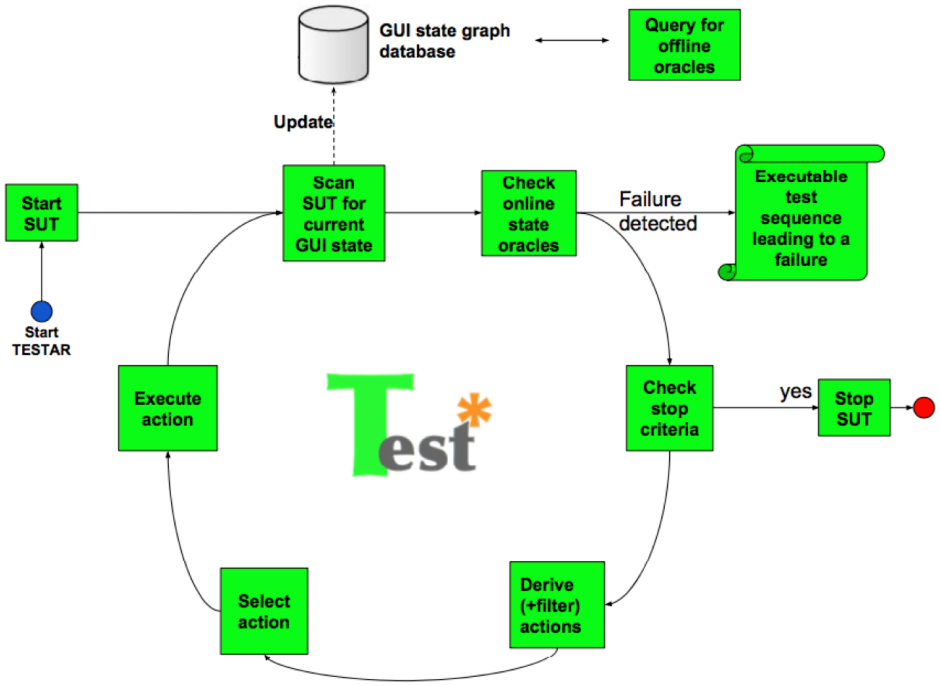

Figure 2: TESTAR test cycle

4.4 The GENERATE-TEST mode

In this mode, the TESTAR tool carries out automated testing following the test cycle depicted in Figure 2.

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

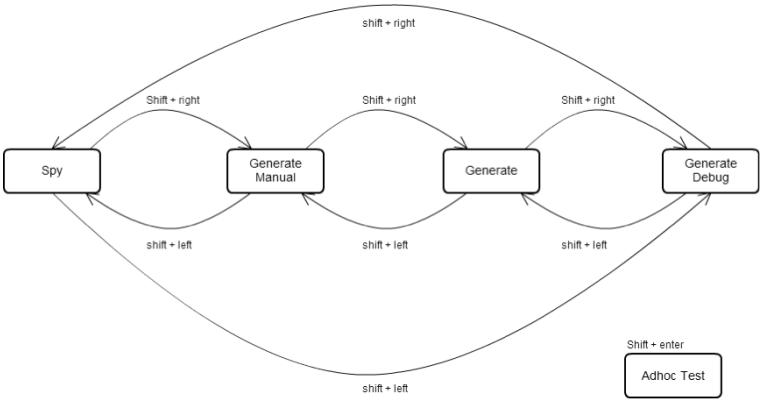

Figure 3: Changing between different modes

- the UI state should not contain any widget with suspicious titles like *error*, *problem*, *exception*, etc.

Now let us do some automated testing. From the SPY mode you can go to the GENERATE modes by hitting Shift + Arrow Right. There are different GENERATE MODES. Figure 3 shows which key combinations will take you from one to the other.

The main working modes that apply for running tests are presented next, and you can switch between them through the keyboard shortcuts Shift + ← (or Shift + →).

- **Generate**. The default operation mode. Runs tests as specified by the test set up.
- **GenerateManual**.You will take over TESTAR 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 UI.
- **GenerateDebug**. Similar to the *Generate* mode, but you will be able to display the UI actions (Shift + 1) and color codes are applied during the test:
  - **green** for UI actions that the test can execute,
  - **red** for the current UI action being executed, and
  - **blue** for UI actions that were already executed.

- **Slow motion**. Hit Shift + Space to (de)activate a delay between the executed UI actions, which will aid to supervise the test execution of more critical UI parts.

☞ **HANDSON**: Run some tests. These are monkey test. TESTAR can *see* the controls of the SUT's GUI and can automatically detect possible actions. It randomly selects and executes these actions. You can interact during the tests at any time, or even stop them through the *panic* keyboard shortcut (Shift + ⇩).

In the "General Settings"-Tab you can configure the number of sequences you want to test and the length (i.e. number of actions) of these sequences. In the "Time Settings"-Tab you can set the action duration and the time that TESTAR waits after executing each action. Play with the settings and observe what happens.

Until now you should have familiarized yourself a bit with the tool. TESTAR does not only generate random sequences, it also records every action it executes and can thus replay sequences. The directory

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

contains all sequences that the tool generated.

☞ **HANDSON**: Set 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.

TESTAR generates more outputs than just the sequences. In Appendix B you can find table that explains what is in the other output directories.

4.5 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 spe-

cific 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[2][3]. 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:

```
C:\testar\bin\output\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.

4.6 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.

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

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

Use the Spy Mode to see whether your action filters have an effect ("Shift + 1" to visualize the actions)!

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 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`.

4.7 The test.settings file

As indicated before, the GUI of TESTAR is basically a dialog that 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 4.13).
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 name directly or through the TESTAR user interface, selecting the desired protocol from the corresponding dropdown-menu in the "General Settings"-Tab.

If you want, you can also edit the files directly. For now, to illustrate we refer to Figure 4, 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?

4.8 Process Listener

Besides looking for expressions of error in the different widgets that make up the interface, TESTAR can interact with the process of desktop applications, listening and finding errors in the buffers of its process. TESTAR works with the output of the processes in two different ways:

- Suspicious Process Output Oracles, responsible for reporting and finishing the execution of an SUT in case of a coindidence.

- Process Logs, responsible for storing other outputs, but without finishing the execution of the SUT.

```
#################################################################
# Oracles based on Suspicious Outputs detected by Process Listeners
#
# (Only available for desktop applications through COMMAND_LINE)
#
# Regular expression defines the suspicious outputs
#################################################################

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

These oracles work and are defined in the same way seen in the section `Design a Test Oracle`, but to enable them we need to activate the Process Listener Enabled mode. Currently these oracles based on process listeners are only enabled for desktop applications executed from the `COMMAND_LINE` option.

```
######################################################################
# 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 4: Part of the test.settings file

```
#################################################################
# Process Logs
#
# Required ProcessListenerEnabled
# (Only available for desktop applications through COMMAND_LINE)
#
# Allow TESTAR to store in its logs other possible matches found in the process
# Use the regular expression .*.* if you want to store all the possible outputs of t
#################################################################

ProcessLogs = .*.*
```

☞ **HANDSON**: Run the calculator with the Spy mode, switch to GenerateManual mode (Shift + →), open the "View" menu and select the option "Digital Grouping". What happened?

Calculator has closed unexpectedly, this is a type of error recognized by TESTAR and therefore the sequence will be stored in the file `bin/output/sequences_unexpectedclose`.

With the process listener disabled, we can only know that the SUT has closed unexpectedly or that something freeze the application. Activate the process listener will help us to obtain more information about the error produced in our application.

☞ **HANDSON**: Run TESTAR and change to the Oracles tab, `Enable Process Listener` and adds `.*Exception.*` as suspect title in the Suspicious Process Output. Run the calculator and repeat the same sequence of actions.

Now the output process information it will be stored according to its sequence in the new folder `bin/output/ProcessLogs`

4.9 Failure BINGO!

☞ **HANDSON**: Now that you know how to use the tool, your task is to setup a longer test, e.g. 30 sequences with a length of 50 actions (feel free to use different values). Run the tool and observe its output! Does it find and report the failures? Can you replay the sequences that found failures and can you reproduce the failures? If the tool executes undesirable actions, improve your setup and restart the test. At the end of this task you should have a folder with several erroneous sequences that can be replayed and expose failures.

☞ **HANDSON**: Find as many failures as possible and report on how to reproduce them. Go to Appendix D there you will find a table and a Bingo card to fill in all the failures you have found and how to reproduce them. Yell **BINGO!!** if you are the first to fill the card. The BINGO is valid if the "How to reproduce fields" can really reproduce the bug. Note that this can be done manually or with TESTAR replay.

4.10 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.

4.11 Play with the Tool and some other applications

The calculator SUT 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!

4.12 Testing web applications

Bitrix24 is a complete suite of social collaboration, communication and management tools for your team. We have already configured a test.settings file for TESTAR for bitrix24 in the web_generic protocol.

Starting up TESTAR up with the web_generic protocol, will use Firefox Browser to start up `www.bitrix24.com/auth`. 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 2. In the next section this is explained.

4.13 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 the default source code that the tool uses to perform its tests. 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 2.

4.13.1 Editing the protocol to logon

If we want TESTAR to automatically login to bitrix24 when we start up the system, we need to edit the web_generic protocol. Bitrix24 remembers the login even if you close the browser during testing, so we can edit `startSystem()` method so that the login is done only once when TESTAR starts. The other option is to edit `beginSequence()` method to login and `finishSequence()` method to logout. If the application would require login each time the browser is closed, then we would have to edit `beginSequence()` method.

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

username: `tvos@pros.upv.es`

password: `testarbitrix24`

First, you need to make a new `CompoundAction` builder, that contains a series of actions that you want to execute when the system starts up. Note that 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. You might need to add some TABs to make sure that this assumption holds.

```
new CompoundAction.Builder()
.add(new Type("my␣user"),0.1) //assume keyboard focus is
                              //on the user field
.add(new KeyDown(KBKeys.VK_TAB),0.5) //assume next focusable
                                     // field is pass
.add(new Type("my␣user␣pass"),0.1)
.add(new KeyDown(KBKeys.VK_ENTER),0.5).build() //assume login
                                               //is performed
                                               //by ENTER
.run(sut, null, 0.1);

return sut;
```

You might need to add some `Util.pause(`*number*`)` to give the website time to render completely before you execute some CompoundAction.

4.13.2 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.

4.14 Testing Java Swing applications

☞ **HANDSON**: Try to test the JEdit application with the generic protocol. What is happening? This is a Java Swing application and the accessibil-

ity 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.

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.

```
#################################################################
# 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 + 1). 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);
    }
//End of Force action
```

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

4.15 Some ways to Analyse TESTAR output

There are several way you can visualize the results of the tests.

- TESTAR view function with screen shots
- Offline graphs created with Graphviz
- Query a OrientDB graph database

4.15.1 Viewing executed test sequences

Sometimes the replay function of TESTAR is not able to reproduce the failure. During testing screenshot are made. These screenshots with action can be viewed in the View mode.

When you start up TESTAR and click on the view button, TESTAR will ask you for the test sequence (file) to inspect as a list of screenshots. With the small reproduction toolbar on top you can navigate through the test. Moving one UI action forward or backward, or jumping to the beginning or end of the test. The current displayed screenshot of the UI state in the test and the UI action that was executed in that state is marked in the screenshot.

HANDSON: Start up TESTAR and click on the View button on the top richt side of the TESTAR GUI. Select one of your previous test sequences. Browse the screen shots by clicking on the previous and next button.

4.15.2 Offline test graphs

TESTAR creates an XML file with information about the executed tests. That XML file can can be transformed into different type of graphs with the graphviz tool (`https://graphviz.org`).

TESTAR represent the executed tests as directed graphs, where: the *nodes* represent UI states and the *edges* or *links* represent the executed UI actions between states.

Different test graphs are generated to view test results from different perspectives (i.e. minimal, tiny, screenshots). These can all be found at `graphs/sequence*/*`.

- **Minimal** (`graph_<timestamp>_minimal.*`) graphs visualize test results with a minimal of information. UI states are represented by rectangles with a number. This number represents a count of the number of times the test traversed through that particular state. UI actions are shown as links between nodes with a number, counting for the number of times that particular action was executed in that state.

  Colors provide additional visual clues about the frequency that states or actions are visited during testing. For example, pink filled circles and links represent UI actions that were derived in the corresponding state but that were never executed. So, for example, the number in the pink links account for the number of unexecuted actions.

- **Tiny** (`graph_<timestamp>_tiny.*`) graphs are an extended version of minimal adding unique state identifiers inside the nodes and actions identifiers inside the links. The order of the actions in the test is indicated by the numbers between brackets.

- **Screenshots** (`graph_<timestamp>_scrshoted.*`) graphs are very useful for analyzing your SUT behaviour. They can for example be used to document the test results. The screenshot are taken whenever the action is performed over a widget on the UI (e.g. left click on a button; the screenshot would be that particular button).

- **Abstract-ed** versions of (`graph_<timestamp>_<type>_abstract.*`) graphs clusters sets of related states and sets of related actions to create a more abstract representation of `<type>` graphs. Here `<type>` can be `minimal, tiny, scrshoted`.

Graphs are stored by default with the .dot[4] extension. To help in

[4]Graph description language: `http://www.graphviz.org/content/dot-language`

their inspection you will need to convert the graphs from .dot to .svg using the `offline_graph_conversion_<timestamp>.bat` script. The SVG files can be visualized using a compatible viewer (e.g. a web browser).

HANDSON: Open the windows *File Explorer* and go to the directory `c:\testar\bin\output\graph`. Select one of the sequences directories. Locate the `offline_graph_conversion_<timestamp>.bat` file and double click on it. The batch script will create the .svg files that will show up in the *File Explorer*. Click on the .svg files to see the graphical output of graphviz in a web browser. With Ctrl +[+] you can zoom in and with Ctrl +[-] you can zoom out. Use the scroll bars of the browser to move up or down, left or right of the graph. Did your selected sequence complete without errors? Can you see when and how a sequence ended in failure?

4.15.3 Using the graph database

Besides the standard graph reports that are available in the `output\graphs` folder, TESTAR has the (EXPERIMENTAL) 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.

A Keyboard shortcuts

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

Keyboard shortcut	**Effect**	**Working modes**		
		Spy	Generate	Replay
Shift + Arrow Down	Close TESTAR immediately (panic button)	✓	✓	✓
Shift + Arrow Left/Right	Switch the working mode	✓	✓	
Shift + Space	Toggle slow motion test		✓	
Shift + 1	Toggle UI actions display	✓	✓	
Shift + 2	Toggle UI widget information display	✓		
Shift + 3	Toggle UI widget extended information display	✓		
Shift + 4	Toggle UI widget-tree display	✓		
Shift + ALT	Toggle widget-tree hierarchy display	✓		
ALT	Define data input values for an UI widget	✓		
CAPS_LOCK/TAB + (Shift) Ctrl	UI widgets' actions filtering	✓		

Table 1: Keyboard shortcuts

B Directories

./settings	Tests set ups
./output	Reports: logs, screenshots, graphs, metrics, serialized tests
./output/temp	Temporary files such as the last recorded test sequence
./output/sequences	All the serialized test sequences
./output/sequences_V	Classified test sequences by verdict V
./output/srcshots	Screenshots of tests UI states and executed UI actions
./output/logs	Tests logging data
./output/graphs	Tests graphing for visual analysis
./output/metrics	Tests performance indicators

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[5]. Check the *StopGenerationOnFault*, *ForceToSequenceLength*, *MaxTime*, *SuspiciousTitles*, *TimeToFreeze* and *ProtocolClass* properties for specific behaviour.

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

[5]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.

D Calculator Failures BINGO!

No.	Type of failure	How to reproduce
1	issue	
2	critical error	
3	strange error error	
4	crash(es)	
5	nullpointer exception	

6	nullpointer exception (simulated)	
7	arithmetic exception	
8	runtime exception	
9	numberformat exception	
10	freeze	

Failure BINGO!

critical error	something else	strange error	arithmetic exception
freeze	wrong calculations	crash	runtime exception
issue	nullpointer exception (simulated)	numberformat exception	crash
bad error message	crash	crash	nullpointer exception

Index

action filter
  clickfilter, 12
  regular expression, 11

clickfilter feature, 12
CompoundAction builder, 18

Dialog Tab, *see* TESTAR Dialog Tabs

filter actions
  clickfilter, 12
  regular expression, 11
FITTEST, 4

general-purpose requirements, 8
GENERATE Mode, 8, 9
  Debug, 9
  Manual, 9
  Slow motion, 10
graph
  database, 24
  graphviz, 23
  offline, 23
GUI, 4

Mode
  GENERATE, 8, 9
    Debug, 9
    Manual, 9
    Slow motion, 10
  REPLAY, 10
  SPY, 7

oracle, 10
  implicit, 10
  suspicious title, 11
orientdb, 24

protocol, 17
  edit, 17
  method
    beginSequence(), 18
    finishSequence(), 18
    get_Verdict(), 18
    startSystem(), 18
  Protocol_desktop_generic.java, 12
  web_generic, 17
protocol_filter.xml, 12

regular expression, 11
REPLAY Mode, 10

SPY mode, 7
sse file, 13
stopping criteria, 8
SUT, 4
  bitrix24, 17
  Calculator (buggy), 4
  connector, 7, 16
    COMMAND_LINE, 7, 16
    SUT_PROCESS_NAME, 16
    SUT_WINDOWS_TITLE, 16
  MS Paint, 17
  Notepad, 17
  VLC media player, 17
Swing applications, 19

Tab, *see* TESTAR Dialog Tabs
test.settings file, 7, 12
TESTAR
  Dialog, 12
  Dialog Tabs
    Filters Settings, 11, 12
    General Settings, 7, 10, 13, 16
    GraphDB, 24
    Oracles, 11
    Time Settings, 10
  Protocol, *see* protocol
  web (www.testar.org), 4

Verdict, 19