TESTAR: AUTOMATED ROBUSTNESS TESTING AT THE USER INTERFACE LEVEL

User Manual

VERSION 1.3 FOR HANDSON DO-IT-YOURSELF PADERBORN (A-TEST @ FSE)

SEPTEMBER 2017

website: www.testar.org

github: https://github.com/TESTARtool/TESTAR

contact: info@testar.org

Contents

List of Figures

List of Tables

Listings

1 About

TESTAR[1] (also written as Test* following the logo of the tool) is a tool for automated testing at the User Interface (UI) level of software applications (e.g. desktop and web applications).

TESTAR is not a Capture-and-Replay tool nor a Visual-based testing tool. TESTAR does not record or need scripts. It uses accessibility technologies to access your User Interface and craft tests on the fly. Whether the User Interface changes so do the tests. This way, no test scripts need to be maintained.

TESTAR v1.0 was developed within the context of the FITTEST[2] (Future Internet Testing) project that run from 2010 till 2013. Since that first version, it went from versions v1.1a to v1.2 due to financing obtained from:

- A proof of concept project financed by the *Universitat Politècnica de València* - Programa de Prueba de Concepto 2014- (SP20141402).
- The SHIP project -SMEs and HEIs in Innovation Partnerships- (EACEA/A2/UHB/CL 554187).
- The PERTEST project -Testing of data persistence and user perspective for new paradigms- (TIN2013-46928-C3-1-R).

Since 2014, TESTAR has been deployed and used in several companies with interesting results, which show its potential to grow into a tool that can help companies improve testing at the User Interface level.

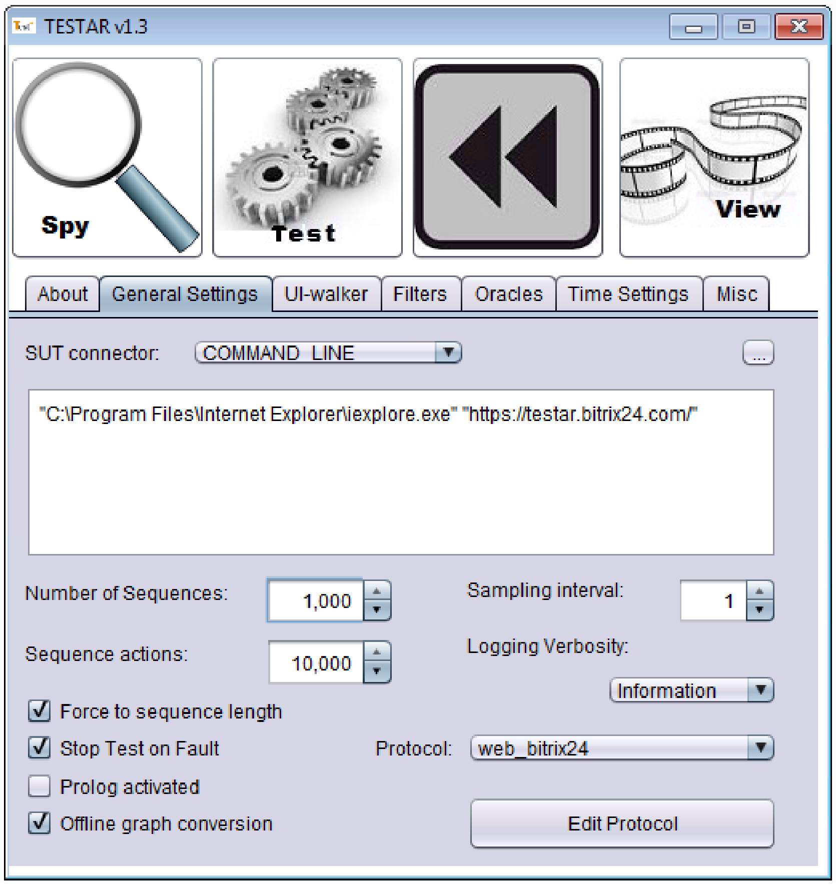
TESTAR is available under the BSD-3 license[3].

Many people have contributed (and are still contributing) to TESTAR. The TESTAR team consists of: Tanja Vos (coordinator of FITTEST and the TESTAR development), Sebastian Bauersfeld (PhD student within the FITTEST project), Urko Rueda (researcher contracted by the FITTEST and SHIP project), Anna Isabel Esparcia (researcher contracted by proof of concept and SHIP project), Francisco Almenar Pedros (Final Master thesis work at the UPV), Mirella-Oreto Martinez Murillo (Final Master thesis work at the UPV), Wouter Cox and Jean Marc (Worked jointly on their bachelor thesis at the OU and developed Linux and Windows 10 platforms support), Davy Kager (bachelor thesis at the OU), Floren de Gier (bachelor thesis at the OU), Fernando Pastor Ricos (bachelor thesis at the UPV).

[1] http://www.testar.org/

[2] http://crest.cs.ucl.ac.uk/fittest/ (EU project no: 257574 FP7 Call 8 ICT-Objective 1.2 Service Architectures and Infrastructures)

[3] http://opensource.org/licenses/BSD-3-Clause



SUT connector: COMMAND_LINE

"C:\Program Files\Internet Explorer\iexplore.exe" "https://testar.bitrix24.com/"

Number of Sequences: 1,000

Sampling interval: 1

Sequence actions: 10,000

Logging Verbosity: Information

☑ Force to sequence length

☑ Stop Test on Fault

☐ Prolog activated

☑ Offline graph conversion

Protocol: web_bitrix24

Figure 1: TESTAR interface

2 Getting TESTAR

During the handson session we will distribute an image of a virtual machine with Windows 10. There are also possibilities (max. 8) to connect with remote desktop to a server and run it on a prepared virtual machine there.

TESTAR can be found in the directory `C:\testar`. You can execute the file:

```
C:\testar\bin\testar.bat
```

to start TESTAR.

3 Running TESTAR

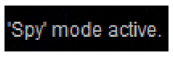
When you start TESTAR you will see its interface as in Figure 1. The 4 big buttons at the top of the tool UI run TESTAR in different modes:

1. **Spy mode** (section 3.2): used to inspect the SUT' User Interface, for example to check that your test set up is ready (check next mode).
2. **Generate mode** (section 3.1): generates and runs automated tests for the established test set up.
3. **Replay mode** (section 3.3): used to replay previously run tests.
4. **View mode** (section 3.4): used to inspect a previously run test at a step-by-step basis. Contrary to the Replay mode, it will not execute the test during inspection. But, you will inspect screenshots of the run test. This mode is key when a test turns out not to be reproducible.

There is an additional **Headless** mode (section 3.5), which is used under batch operations without human intervention.

The different tabs are to manipulate most of the settings properties that are all summarized in Appendix D and discussed in Section **??**.

You are able to switch between Spy en Generate mode by using the Shift + ← or Shift + → keyboard shortcuts[4]. You will see a flashing message in the screen indicating the currently running working mode. For example:


'Spy' mode active.

Additionally, you can stop TESTAR at any time during Spy, Generate and Replay modes by pressing the Shift + ↓ keyboard shortcut. This is particularly usefull as a ***panic button*** during test and replay modes, which immediately stops any undesired activity being performed (e.g. erasing files, printing documents). You can prevent such undesired scenarios with a correct test set up (see section **??**).

3.1 Generate mode

In the generate 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 fault 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 fault has been found or until a stopping criteria is reached.

With the right test set up all you will 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: like that 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.

You can interact during the tests at any time, or even stop them through the *panic* keyboard shortcut (Shift + ↓). The main working modes that apply for the running tests are presented next, and you can switch between them through the keyboard shortcuts Shift + ← (or Shift + →).

[4]Check appendix E for additional information

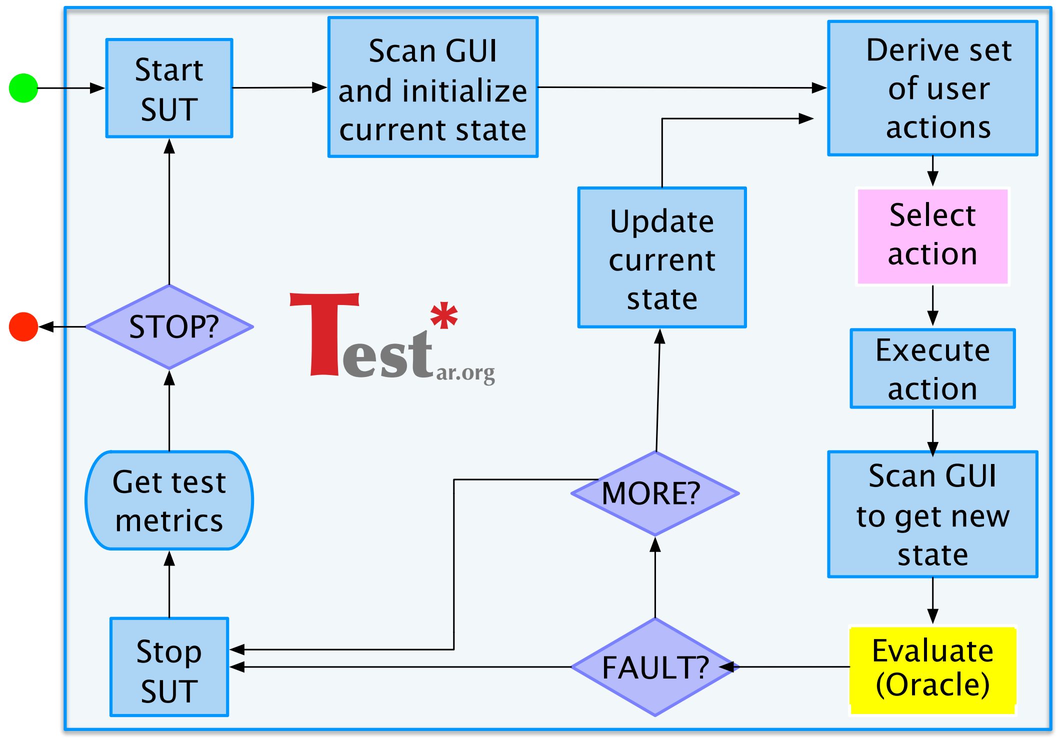


Figure 2: TESTAR testing cycle

- **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 colour codes are applied during the test:

  green for UI actions that the test can execute,

  red for the current UI action being executed, and

  (alpha) 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.

All tests data generated by a TESTAR execution is stored inside the *output* folder. There is:

- ***.log**: tool logs
- ***.dbg.log**: STDOUT logs

- **sequences/***: executed tests
- **sequences_VERDICT/***: classified tests
- **logs/*.log**: tests logs
- **logs/*_clusters.log**: tests clustering logs
- **logs/*_curve.log**: tests UI exploration curve logs
- **logs/*_stats.log**: tests statistics logs
- **logs/*_testable.log**: tests table (ordered executed UI actions) logs
- **scrshots/***: tests UI states and actions screenshots
- **graphs/***: tests graphing data
- **metrics/*.csv**: tests metrics

Details about the data and how to use them in in Section 5.

3.2 Spy mode

Spy mode enables the tester to inspect the User Interface of the SUT. This provides useful information to prepare for your test set up. Once you press the *Spy* button, your SUT will start. You can interact with your SUT in the usual way, but the TESTAR Spy will provide the following capabilities:

1. **UI actions**: To display the UI actions which will be available during tests, press keyboard shortcut: Shift + 1 . Figure 3 shows an example for the *calculator* SUT, where the green dots mark left click actions. Other types of actions might be displayed with different visual appearances as described in table 1.

Green dot	Left click
Green circle (size small)	Right click
Green circle (size medium)	Left double click
Green circle (size big)	Combined Left click and Right arrow
Green Text	Type text (combined click, remove text and type text)
Green Arrow	Drag & Drop Operations and Slides
Green Line (at widget bottom)	Move mouse pointer to widget

Table 1: Spy mode through Shift + 1 - Actions visual appearances

2. **Basic widget information**: Shift + 2 enables/disables additional information about the UI widgets when you hover over them. Figure 4 shows an example for the *calculator* SUT. The widget under the mouse pointer is highlighted with a yellow overlay and a brief overlay panel showing four properties.

3. **Extended widget information**: Shift + 3 enables/disables extended information about the UI widget under the mouse pointer. Figure 5 shows an example for the *calculator* SUT, where additional widget properties are displayed in an overlay panel.

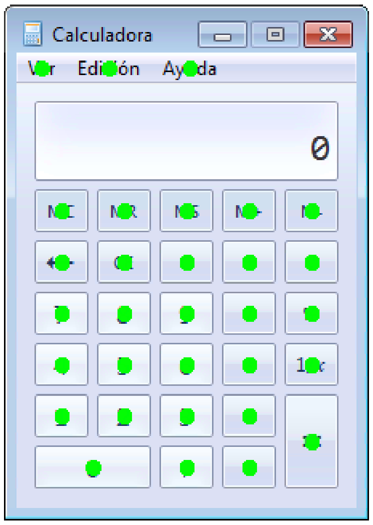
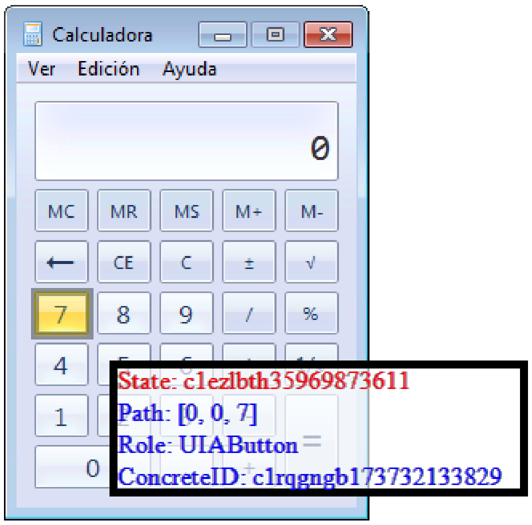
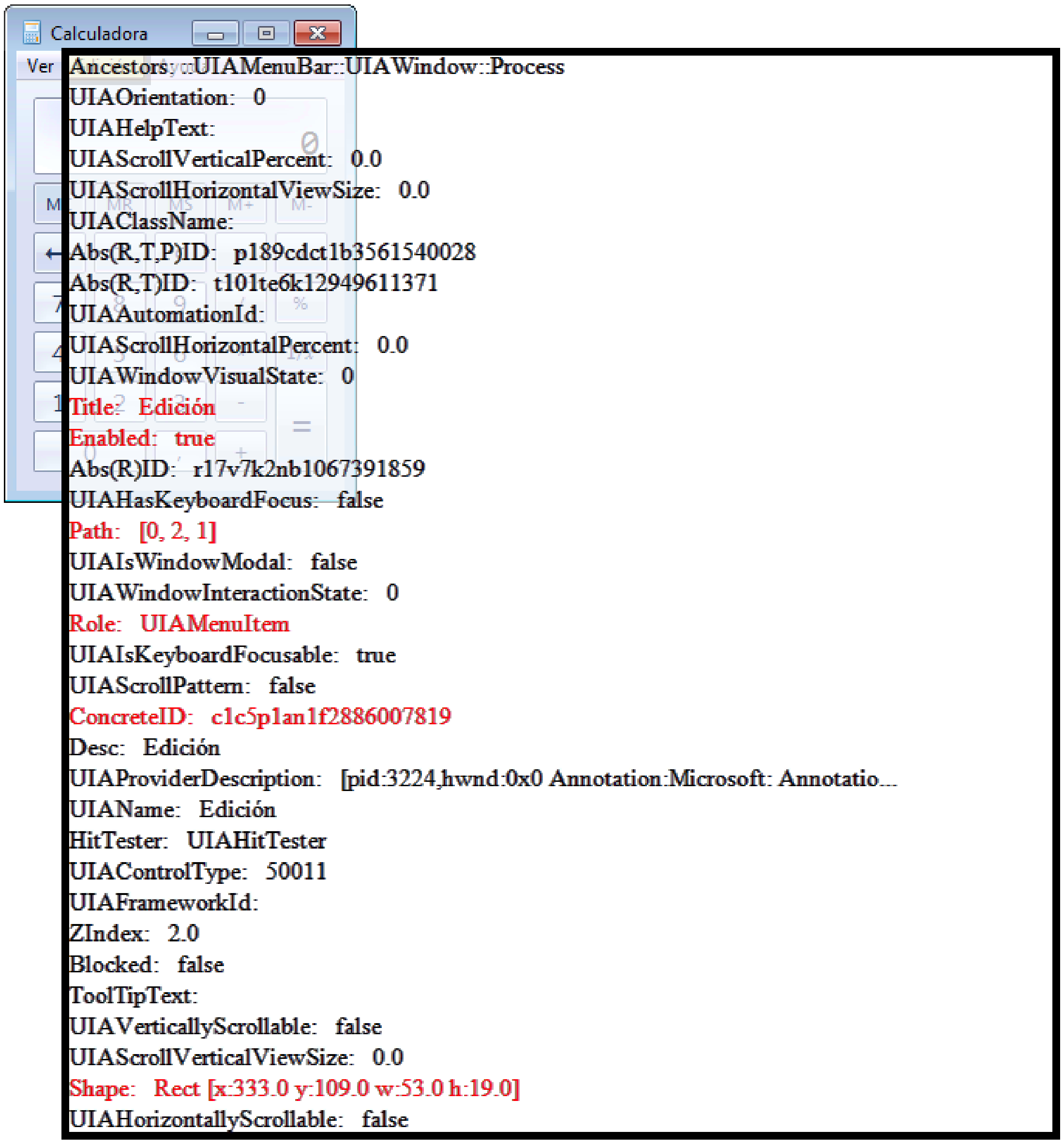


Figure 3: Spy mode: showing UI actions as green dots



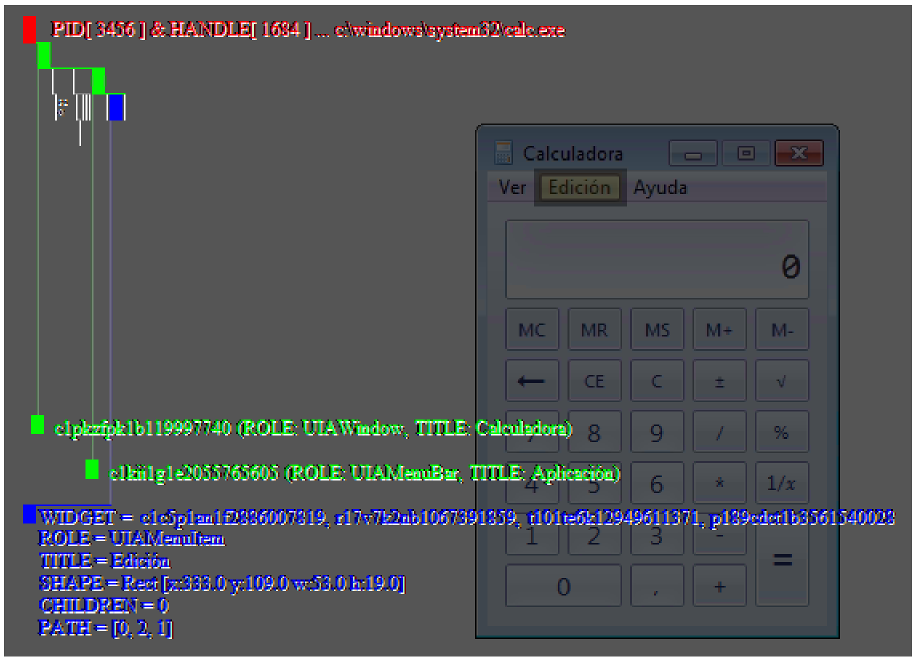
State: c1ezlbth35969873611
Path: [0, 0, 7]
Role: UIAButton
ConcreteID: c1rqgngb173732133829

Figure 4: Spy mode showing basic widget information.



```
Ancestors: ::UIAMenuBar::UIAWindow::Process
UIAOrientation:  0
UIAHelpText:
UIAScrollVerticalPercent:  0.0
UIAScrollHorizontalViewSize:  0.0
UIAClassName:
Abs(R,T,P)ID:  p189cdct1b3561540028
Abs(R,T)ID:  t101te6k12949611371
UIAAutomationId:
UIAScrollHorizontalPercent:  0.0
UIAWindowVisualState:  0
Title:  Edición
Enabled:  true
Abs(R)ID:  r17v7k2nb1067391859
UIAHasKeyboardFocus:  false
Path:  [0, 2, 1]
UIAIsWindowModal:  false
UIAWindowInteractionState:  0
Role:  UIAMenuItem
UIAIsKeyboardFocusable:  true
UIAScrollPattern:  false
ConcreteID:  c1c5p1an1f2886007819
Desc:  Edición
UIAProviderDescription:  [pid:3224,hwnd:0x0 Annotation:Microsoft: Annotatio...
UIAName:  Edición
HitTester:  UIAHitTester
UIAControlType:  50011
UIAFrameworkId:
ZIndex:  2.0
Blocked:  false
ToolTipText:
UIAVerticallyScrollable:  false
UIAScrollVerticalViewSize:  0.0
Shape:  Rect [x:333.0 y:109.0 w:53.0 h:19.0]
UIAHorizontallyScrollable:  false
```

Figure 5: Spy mode showing extended widget information.



```
PID[ 3456 ] & HANDLE[ 1684 ] ... c:\windows\system32\calc.exe
c1pkzfpk1b119997740 (ROLE: UIAWindow, TITLE: Calculadora)
c1kii1g1e2055765605 (ROLE: UIAMenuBar, TITLE: Aplicación)
WIDGET = c1c5p1an1f2886007819, r17v7k2nb1067391859, t101te6k12949611371, p189cdct1b3561540028
ROLE = UIAMenuItem
TITLE = Edición
SHAPE = Rect [x:333.0 y:109.0 w:53.0 h:19.0]
CHILDREN = 0
PATH = [0, 2, 1]
```

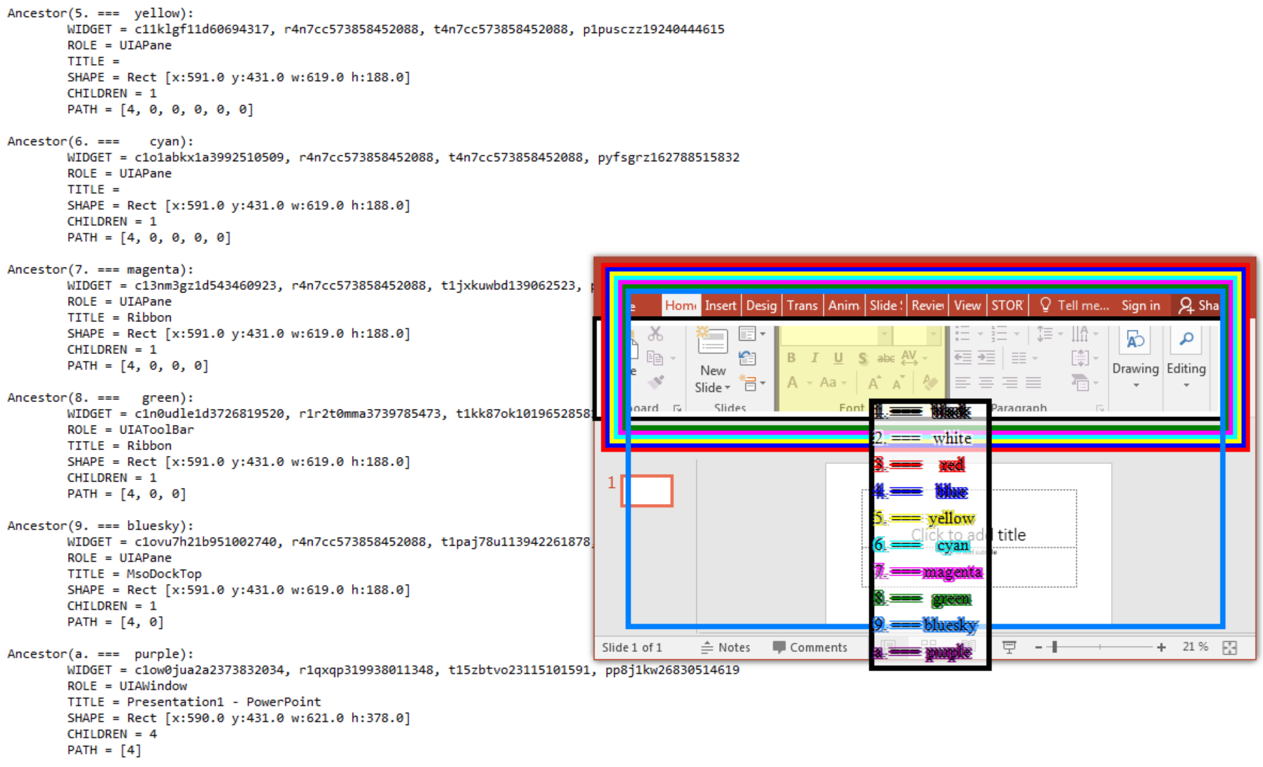
Figure 6: Spy mode showing the whole widget tree.

4. **Graphical widget-tree**: Shift + 4 enables/disables the information about the UI widget-tree through a graphical display[5]. Figure 6 shows an example for the *calculator* SUT, where the widget under mouse pointer is represented by a blue box, its ancestor widgets in the widget-tree as green boxes in the upper positions and sibling widgets represented by short grey vertical lines (same height as boxes). The top red box represents the SUT process and the first green box from top the current displayed window of the SUT UI. The bottom part provides widgets properties for the green and blue boxes. This graphical display is particularly interesting to be aware of hidden widgets in the SUT UI, for example containers (e.g. a panel).

5. **Widget-tree hierarchy**: Shift + ALT enables/disables the display of ancestor widgets in the widget-tree of the widget that is hovered over. Figure 7 shows an example for the *Powerpoint* SUT, where different overlay colored rectangles are used to mark the corresponding ancestors. A legend of the used colors is provided, from top to bottom the first and last ancestors. You will also find detailed ancestors information at the standard output (e.g. console), as shown in the left part of the Figure. This can be complemented with the graphical widget-tree display from previous point.

6. **Data input values**: ALT can be used to define some input values for the widget under mouse pointer. Figure 8 shows an example, where a popup dialog allows to establish the type of input values that should be used for the corresponding widget. The popup list displayed can be customised with as many types of input values as required for the SUT. To do so, edit the file `input\_values.xml`[6] and append as

[5]Rows denote depth levels in the widget-tree hierarchy and columns widget siblings
[6]This file can be found at TESTAR's root installation folder



```
Ancestor(5. ===  yellow):
        WIDGET = c11klgf11d60694317, r4n7cc573858452088, t4n7cc573858452088, p1pusczz19240444615
        ROLE = UIAPane
        TITLE =
        SHAPE = Rect [x:591.0 y:431.0 w:619.0 h:188.0]
        CHILDREN = 1
        PATH = [4, 0, 0, 0, 0, 0]

Ancestor(6. ===    cyan):
        WIDGET = c1o1abkx1a3992510509, r4n7cc573858452088, t4n7cc573858452088, pyfsgrz162788515832
        ROLE = UIAPane
        TITLE =
        SHAPE = Rect [x:591.0 y:431.0 w:619.0 h:188.0]
        CHILDREN = 1
        PATH = [4, 0, 0, 0, 0]

Ancestor(7. === magenta):
        WIDGET = c13nm3gz1d543460923, r4n7cc573858452088, t1jxkuwbd139062523, p
        ROLE = UIAPane
        TITLE = Ribbon
        SHAPE = Rect [x:591.0 y:431.0 w:619.0 h:188.0]
        CHILDREN = 1
        PATH = [4, 0, 0, 0]

Ancestor(8. ===   green):
        WIDGET = c1n0udle1d3726819520, r1r2t0mma3739785473, t1kk87ok10196528581
        ROLE = UIAToolBar
        TITLE = Ribbon
        SHAPE = Rect [x:591.0 y:431.0 w:619.0 h:188.0]
        CHILDREN = 1
        PATH = [4, 0, 0]

Ancestor(9. === bluesky):
        WIDGET = c1ovu7h21b951002740, r4n7cc573858452088, t1paj78u113942261878
        ROLE = UIAPane
        TITLE = MsoDockTop
        SHAPE = Rect [x:591.0 y:431.0 w:619.0 h:188.0]
        CHILDREN = 1
        PATH = [4, 0]

Ancestor(a. ===  purple):
        WIDGET = c1ow0jua2a2373832034, r1qxqp319938011348, t15zbtvo23115101591, pp8j1kw26830514619
        ROLE = UIAWindow
        TITLE = Presentation1 - PowerPoint
        SHAPE = Rect [x:590.0 y:431.0 w:621.0 h:378.0]
        CHILDREN = 4
        PATH = [4]
```

Figure 7: Spy mode showing the widget-tree hierarchy

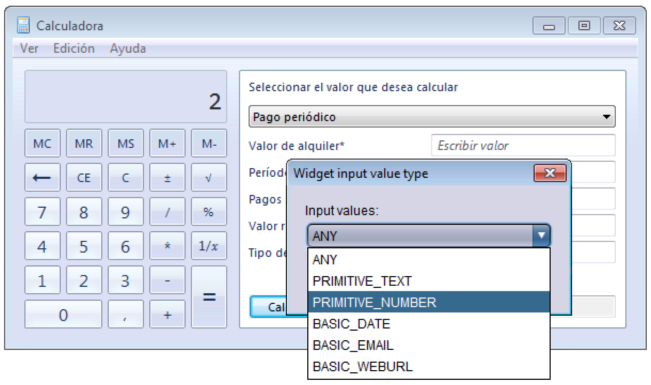


Figure 8: Spy mode, defining input values through ALT

many input values as you require for your SUT. In listing 1 you can find an example.

Listing 1: input_values.xml

```xml
<?xml version="1.0"?>
<TESTAR_inputvalues version="1.0.20170322">
  <data_types> <!-- note: negative data types are reserved -->
    <!-- <data type="1" desc="data_type_description" example="example_value"/> -->
  </data_types>
  <input_values> <!-- provide a value or list of values for each declared data
      type -->
    <!-- <input type="1" value="foo_value"/>
    <input type="1" value="boo_value"/> -->
  </input_values>
</TESTAR_inputvalues>
```

7. **UI actions filters:** CAPS_LOCK enables/disables a feature that enables you to filter actions directly 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 hit Ctrl to filter all actions on this widget from being selected during testing. You can hit Shift + Ctrl to undo the filtering. 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 root installation folder.

3.3 Replay mode

TESTAR will ask you for the test sequence (file) to replay. Replaying a saved test implies to run your SUT and execute the UI actions in the very same order and with the same delays that were used in the run tests. You can switch to a **ReplayDebug** working mode (Shift + →), which displays the current executed UI action (red color). Additionally, you can slow down the replay through the Shift + Space keyboard shortcut.

However, your tests might become not reproducible if the SUT behaviour is not stable for the same tests. In that case, you can look into the next mode (View), which has the main difference of not re-executing the tests, but displaying the list of states and actions that conform your saved test sequence.

Yet, the SUT might be starting from a different state. Did you considered deleting/restoring SUT specific settings files in your test set up?

3.4 View mode

TESTAR will ask you for the test sequence (file) to inspect as a list of screenshots. A tin reproduction toolbar will allow you to 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 is the UI state in the test, and the UI action that was executed in that state is remarked in the screenshot.

3.5 Headless mode

TESTAR can be executed without its user interface, which enables to automate tests executions from scripts. To do so you will need to edit the file *test.settings* and set the property *ShowVisualSettingsDialogOnStartup* to the value *false*. Whether you want to see TESTAR user interface again then switch the value to *true*, for example to adjusting the different settings for your tests through the user interface (or you can simply edit the properties' values directly in the *test.settings* file). Please, make sure that the property *Mode* is set to the value *Generate*, which will be used for running your tests.

Alternatively, you can simply execute the *batchrun.bat* batch script[7]. A sample execution would be: *batchrun.bat 10*, where tests will be repeated 10 times. For example, if your test set up is defined to run 3 test sequences of 1000 UI actions each, then your batch execution would repeat the 3 test sequences, of 1000 UI actions, 10 times (30 tests with 30000 UI actions in total).

Additionally, you can bypass some properties of the *test.settings* through the main run script *run.bat*, which accepts several arguments:

- **-Dheadless=true/false**. Run with (false) or without (true) the user interface.
- **-DTG=random/qlearning/qlearning+/maxcoverage/prolog/evolutionary**. Sets the UI action selector (algorithm) to use during tests.
- **-DSL=positive_number**. Sets the test sequences length by the number of UI actions.
- **-DGRA=true/false**. Enables/disables the graph resuming feature for future tests (resumes from a previous test graph).
- **-DF2SL=true/false**. Continues (true) testing in the presence of a SUT FAIL.
- **-DTT=positive_number**. For the action selector algorithms, sets the number of UI typing actions, with different texts, that must be performed over the same state' widget to treat the action as executed.
- **-DSST=0.0 .. 1.0**. It activates the widget-tree's build cache, which will speed up the tests. Lower values will get better performance, but the widget-trees can easily get out of sync with the SUT user interface. A recommended value is around 0.95. It is also suggested to use this option with caution and to not use it at all for safe testing.
- **-DUT=true/false**. Activates (true) unattended tests, which will disable all the user events for TESTAR. It will guarantee a more reliable tests due the event handling will not interfere with the tests. However, it will also disable key keyboard shortcuts like the *panic button*.

[7]Under Windows environments. You might set a similar script for other environments like Linux

4 Test protocol

TESTAR provides a customizable test protocol as a Java source file[8], which is automatically compiled by the tool by user request (*Save and Compile* button). The builtin editor of the Java protocol contains three components: 1) top: the source code of the test protocol, 2) middle: a button "*Save and Compile*", which saves and compiles the protocol and 3) bottom: the *Error Console*, which informs you about potential errors during compilation.

The protocol is depicted in figure 9, which contains a set of Java methods[9]. Find details about them in the next subsections. Their signature is presented in listing 2.

Listing 2: The Java Methods that can be customized

```java
// initial setup before starting SUT test
void initialize(Settings settings)

// determines whether to continue SUT testing (additional runs)
boolean moreSequences()

// any action to be taken during SUT start
SUT startSystem()

// clean-up tasks for new test runs
void beginSequence()

// step-by-step STATE of the SUT, with an attached ORACLE
State getState(SUT system)

// determines the STATE ORACLE verdict
Verdict getVerdict(State state)

// determines the stopping criteria
boolean moreActions(State state)

// the set of available ACTIONs from a SUT's STATE
Set<Action> deriveActions(SUT system, State state)

// which ACTION should be PERFORMED next (i.e. random, Search-Based)
Action selectAction(State state, Set<Action> actions)

// runs an ACTION from a SUT STATE, with return code (success?)
boolean executeAction(SUT system, State state, Action action)

// any action to be taken during SUT stop
void stopSystem(SUT sut)

// finishing tasks for an ending test run
void finishSequence(File recordedSequence)
```

[8]To edit it refer to section **??**.

[9]There is no Java method for the *FAULT* checker in the figure

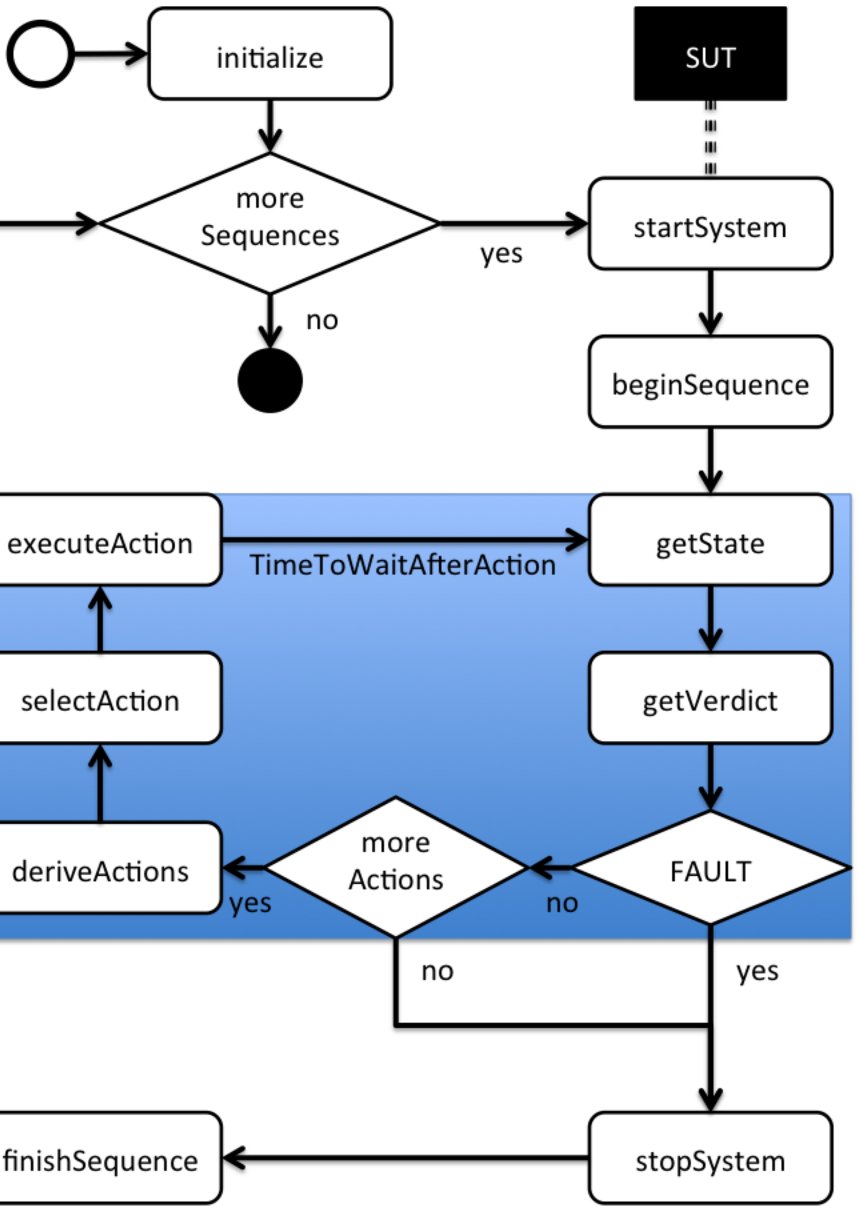


Figure 9: Test protocol

4.1 *initialize* method

Signature of the method: *void initialize(Settings settings).*

Grants the opportunity to perform any initialization tasks before the tests are started. You may want to scan the properties of your *test.settings* to achieve particular goals.

4.2 *moreSequences* method

Signature of the method: *boolean moreSequences().*

Perhaps you want to override the default checker that has been defined by your test set up and introduce a dynamic decision based on particular criteria. For example, you could have instrumented your SUT to measure the code coverage achieved by the tests. Then, you can continue with extra test sequences until a succesful coverage threshold is satisfied.

4.3 *startSystem* method

Signature of the method: *SUT startSystem().*

Perform any tasks you require for your tests around the SUT start process. For example, in web applications it is common the necessity of performing a user login action. Lets assume that your web application displays a login page right after starting it. Then, to automate a user login the method can be populated with the next code:

```
SUT sut = super.startSystem();

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 adjust the combination of user events (e.g. left clicks on text boxes, pressing keys, typing texts) to achieve your desired goal. You might also need to scan the UI of your SUT (section 4.5) for a better decision on the user events to perform in the proper UI widgets.

Furthermore, you can activate your SUT in this method by its 1. UI window title or its 2. process name:

1. return super.startSystem("SUT_WINDOW_TITLE:my SUT window title");
2. return super.startSystem("SUT_PROCESS_NAME:my SUT process name");

4.4 *beginSequence* method

Signature of the method: *void beginSequence().*

You can perform any tasks you require for the next test sequence to work as you expect. For example, do you need to set your database in a concrete state?

It is also important for the reproducibility of your recorded tests (check section 3.3) to put your SUT in a concrete starting state. The large majority of SUTs remembers specific settings or saves the position of its windows as they have been during the last session. If you do not restore the SUT's settings to their defaults, a previously recorded test sequence might not be replayed properly. For example, because the SUT starts in a different UI state (e.g. the UI starts with the last edited document).

4.5 *getState* method

Signature of the method: *State getState(SUT system)*.

Here, the widget-tree (state) of your current SUT UI is built. TESTAR queries the UI through assistive technologies, for example the underlying Operating System's Accessibility API, which has the capability to detect and expose UI widgets and their properties (e.g. role, title, shape).

Are you interested in a particular state of your UI? You have the chance to scan the current widget-tree of your SUT and perform any tasks to enhance your test. For example, you might alter your database if the UI displays a table and you are not satisfied with a table containing few rows. For a good test you could expect the table to contain hundreds of rows (or more) and so, you can populate your database with additional data to retrieve a bigger table.

4.6 *getVerdict* method

Signature of the method: *Verdict getVerdict(State state)*.

Besides the default oracles (e.g. crashes, freezes and suspicious titles) provided by TESTAR, you can codify more advanced oracles through incremental build of SUT requirements. For example, the next code listing codifies an oracle to check for readability of your UI where all the texts should have a minimum font size to be legible:

```
Verdict verdict = super.getVerdict(state);
// MORE SOPHISTICATED ORACLES CAN BE PROGRAMMED HERE (the sky is the limit ;-)
Role role; Shape shape;
for(Widget w : state){ // iterate over all the current UI widgets
   role = w.get(Tags.Role, null); shape = w.get(Tags.Shape, null);
   verdict = verdict.join(getSmallTextVerdict(state, w, role, shape));
}
return verdict;
--
private 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 != null && shape.height() < MINIMUM_FONT_SIZE)
        return new Verdict(Verdict.SEVERITY_WARNING, "Not all texts have a size >=
            " + MINIMUM_FONT_SIZE + "px", new ShapeVisualizer(BluePen,
            w.get(Tags.Shape), "Too small text", 0.5, 0.5));
   else return Verdict.OK;
}
```

4.7 *moreActions* method

Signature of the method: *boolean moreActions(State state)*.

You can override your test set up and decide on whether to continue the test sequence. You might want to code your goal across other methods of the protocol, for example to know about the UI states that the test did traverse. Are you interested to test the state X? Was the state X tested?

4.8 *deriveActions* method

Signature of the method: *Set<Action> deriveActions(SUT system, State state).*

This is a critical method of the protocol where you can decide on the feasible UI actions for your tests. Next code listing builds left clicks, typing random[10] texts and sliding scrollable components over widgets that are enabled, not blocked by other widgets (e.g. in top), white-listed, not black-listed and which are clickable, type-able and slide-able.

```
Set<Action> actions = super.deriveActions(system,state);

StdActionCompiler ac = new AnnotatingActionCompiler(); Drag[] drags;
for(Widget w : state){ // iterate through all widgets
   // enabled and non-blocked widgets
   if (w.get(Enabled, true) && !w.get(Blocked, false)) continue;
   // taboo widgets (UI actions filters from the Spy mode section)
   if (blackListed(w)) continue;
   // left clicks
   if (whiteListed(w) || isClickable(w)) actions.add(ac.leftClickAt(w));
   // type into text boxes
   if (whiteListed(w) || isTypeable(w)) actions.add(ac.clickTypeInto(w,
      this.getRandomText(w)));
   if ((drags = w.scrollDrags(SCROLL_ARROW_SIZE,SCROLL_THICK)) != null){
      for (Drag drag : drags){ // sliding actions
         actions.add(ac.dragFromTo(
            new AbsolutePosition(Point.from(drag.getFromX(),drag.getFromY())),
            new AbsolutePosition(Point.from(drag.getToX(),drag.getToY()))
         ));
      }
   }
}

return actions;
```

You might want to customise your own UI actions, for example with specific texts to type into text boxes, right clicks, drag and drops, mouse gestures, etc.

4.9 *selectAction* method

Signature of the method: *Action selectAction(State state, Set<Action> actions).*

The default behaviour is to select a UI action from the list based on the algorithm that has been defined by the test set up (e.g. random, qlearning). However, you can interfere in the next UI action to be executed by scripting your own logic. You could even build/select an action outside the list. For example, did you identify a UI state which is not interesting

[10]From feasible input values as defined by the *input_values.xml* specification (check section 3.2)

at all for your test and you prefer to execute a concrete action (e.g. press ESC key)? Then, simply build and select the ESC action when you are at that particular UI state.

4.10 *executeAction* method

Signature of the method: *boolean executeAction(SUT system, State state, Action action).*

Here, the selected UI action will be executed into your SUT. You might want to perform additional tasks (e.g. check a database consistency after the action) or even check the status of action execution (was it successful?).

4.11 *stopSystem* method

Signature of the method: *void stopSystem(SUT sut).*

Perform any tasks that suit during the SUT shutdown at the test sequence end. For example, check that your SUT has indeed shutdown correctly, clean any files that were created during the test or clean your database to a default state.

4.12 *finishSequence* method

Signature of the method: *void finishSequence(File recordedSequence).*

Place here the code you need to follow up the test sequence finalisation. For example, you could start your own reporting scripts for the executed test.

5 Test reports

All tests data generated by a TESTAR execution is stored inside the *output* folder:

- ***.log**: tool logs
- ***.dbg.log**: STDOUT logs
- **sequences/***: executed tests
- **sequences_VERDICT/***: classified tests
- **logs/*.log**: tests logs
- **logs/*_clusters.log**: tests clustering logs
- **logs/*_curve.log**: tests UI exploration curve logs
- **logs/*_stats.log**: tests statistics logs
- **logs/*_testable.log**: tests table (ordered executed UI actions) logs
- **scrshots/***: tests UI states and actions screenshots
- **graphs/***: tests graphing data
- **metrics/*.csv**: tests metrics

Check the next sections to understand the data, and how to analyse it (section 5.7).

5.1 Tool logs

Every time you start tests in TESTAR you should see a time stamped log file (.log) and, if *powershell* utility was successfully used by the *run.bat* script, you should also see a time stamped debug log file (.dbg.log). The former, will indicate the timestamp of TESTAR start, the used settings and the working operational mode. The latter, will store all the STDOUT during tests execution. Simplified and commented samples are presented in the next listings[11]:

```
05.July.2017 13:02:59 TESTAR v1.3 is running with the next settings:
-- settings start ... --
... // list of pairs <property,value>
-- ... settings end --
'Generate' mode active. // working operational mode being used for the tests
```

```
>"JVM_HOME\bin\java.exe" -ea -server -Xmx1g -jar testar.jar // command used to run
Test settings is <./settings/desktop_wincalc/test.settings> // settings being used
// input_values.xml contents
// libraries loading
<Q-Learning> test generator enabled (parameters) // algorithm
[START] Running processes (12): // a list with process PID, HANDLE and name
SUT is running after <0> ms ... waiting UI to be accessible
```

[11]Using default logging (check *LogLevel* in section D)

```
SUT is running after <509> ms ... waiting UI to be accessible
SUT accessible after <612> ms
// performance metrics per action (as many lines as the test sequence length)
Finish sequence
[END] Running processes (12): // a list with process PID, HANDLE and name
currentseq: .\output\temp\tmpsequence // temporal Java serialization of the test
// test metrics
// other tool messages
```

5.2 Tests sequences

All the tests you have executed will be serialized as Java objects into the *sequences/* folder. Additionally, they are classified and copied into several *sequences_ VERDICT/* folders, as many as different verdicts that were gathered on finished tests. For example, you might find a *sequences_ OK* folder if almost one of the test sequences did PASS and a *sequences_ suspicioustitle* folder if almost one of the test sequences contained a suspicious title in the UI (e.g. An unexpected error ...).

You will be able to use these serialized files to Replay (section 3.3) and View (section 3.4) your tests.

5.3 Tests logs

Test logs contains detailed information about your tests. For each test sequence you will find the next set of logs:

- **Log** (logs/sequence*.log). A detailed list of ordered executed test UI actions. A simplified and commented sample is provided in listing 3.
- **Clusters** (logs/*_clusters.log). States and actions clusters. A simplified and commented sample is presented in listing 4.
- **UI exploration curve** (logs/*_curve.log). Metrics per action. A simplified and commented sample is displayed in listing 5.
- **Statistics** (logs/*_stats.log). Summary statistics. A simplified and commented sample[12] is illustrated in listing 6.
- **Actions table** (logs/*_testable.log). A detailed list of ordered test UI actions. A simplified and commented sample is shown in listing 7.

Listing 3: Sample test log

```
05.July.2017 13:03:00 Starting SUT ... // timestamp of the test start
Starting sequence 1 (output as: sequence1) // number of test sequence and folder

// repeated for each test sequence UI action

// action and state identifiers
```

[12]Data was obtained by a new test resuming from the test at listing 5

```
Executed [action_number]: action = ACs3xabg802043572381 (AAs3xabg802043572381)
    @state = SC1160ylw3862984133231 (SR4ygmlp3761868872537)
   SUT_KB = 12800, SUT_ms = 16 x 0 x 13.68% // SUT RAM and CPU (user x system) usage
   ROLE = LeftClickAt // action ROLE
   TARGET = // action's UI target widget
      // target widget identifiers
      WIDGET = WC1pno85v21350358828, WR1h6ibpb92248851252, WT1m8m97k131880447283,
          WPpcrzut1d619024393
      ROLE = UIAButton // widget ROLE
      TITLE = Porcentaje // widget TITLE
      SHAPE = Rect [x:684.0 y:454.0 w:34.0 h:27.0] // widget SHAPE
      CHILDREN = 0 // widget's children
      PATH = [0, 0, 30] // widget path in the widget-tree
   DESCRIPTION = Left Click at 'Porcentaje' // action DESCRIPTION
   TEXT = Compound Action = // action as TEXT
      Move mouse to (701.0, 467.5). // screen coordinates
      Press Mouse Button BUTTON1 // left button
      Release Mouse Button BUTTON1

// test finish messages
Shutting down the SUT...
Sequence 1 finished.
Copying generated sequence ("sequence1") to output directory...
Copying classified sequence ("sequence1") to sequences_ok folder...
TESTAR stopped execution at 05.July.2017 13:03:10
Test duration was 10 seconds or 0 minutes or 0 hours
```

Listing 4: Sample test clusters

```
STATES CLUSTERS: // clustering based on Abstract (ROLE) and Concrete identifiers
[state_cluster_number] SR4ygmlp3761868872537 contains:
   (1) SC1160ylw3862984133231 (2) SCe23kmz3881464978174 (3) SC1vq8pzz388129504854
        (4) SC163rqiq3873367995483 (5) SCnsin3r3893728607306 (6)
        SCf4pb57387250135947 (7) SC11u5yt13871143516693 (8) SCfvnoc3388574132644
   (9) SCx32qhb387560692598 (10) SC82s3jw3881175443024

ACTIONS CLUSTERS: // clustering based on Abstract and Concrete identifiers
[action_cluster_number] AAdqo8ce7f949894849 contains:
   (1) ACdqo8ce7f949894849
```

Listing 5: Sample test UI exploration curve

```
________________UNIQUE ________ABSTRACT ___TOTAL
  #, states, actions, states, actions, unq, abs, unx, maxpath, minCvg, maxCvg,  KCVG
// # = action order, unq = UNIQUE/concrete, abs = ABSTRACT, unx = unexplored actions
  0,      1,       0,      1,       0,   1,   1,   0,       2,     0%,     0%,   0@0
  1,      1,       1,      1,       1,   2,   2,  32,       2,     3%,     3%,  3@33
  2,      1,       2,      1,       2,   3,   3,  31,       2,     6%,     6%,  6@33
 ...
 20,      8,      20,      1,      20,  28,  21, 244,       6,     3%,    18%, 7@26a
 ...
 40,     16,      38,      4,      37,  54,  41, 453,      10,     2%,     99%, 7@49a
 ...
 50,     18,      48,      4,      47,  66,  51, 569,      12,     2%,     99%, 7@61a
// KCVG = coverage @ total derived actions (a = x10, b = x100, c= x1000, ...)
```

Listing 6: Sample test statistics

```
// unexplored state = there are pending UI actions (derived) to be executed
total states, unique states, abstract states, unexplored states
         134,            44,               9,                43
// unexplored action = not executed (check -DTT parameter - headless working mode).
total actions, unique actions, abstract actions, unexplored actions
          100,             98,               97,               1413
total unique, total abstract, maxpath, minCvg, maxCvg, VERDICT
         142,            106,      19,  2.38%, 99.90%,    PASS

=== GRAPH RESUMING ===
Known states: 18 // states discovered from previous test sequence
Revisited states: 12 // revisited states from previous test sequence
New states: 26 // new states not discovered at previous test sequence

=== TEST GENERATOR ===
Name: random
```

Listing 7: Sample test actions table

```
ACTION_TYPES: T = ClickTypeInto, RC = RightClickAt, LC = LeftClickAt // codes for action types in the test set up
// table will contain as many lines as UI actions in the test sequence
 #, RAM(KB), CPUser(ms), CPUsys(ms), CPU(%),                  FROM, x,                    TO, x,         // table header
                ACTION, x, ACTION_TYPE, ( (                WIDGET,           ROLE,             TITLE ) [ parameter* ] )+
// # = UI action order
 1,  12920,          0,         15,  10.34, SCgktqw33c43868414317, 7, SCgktqw33c43868414317, 7,
   ACxc9mo17f377244434, 1,          LC,    (  WCphm7bz181224389293,      UIAButton,                  0 ) [            ]
// FROM = state identifier at which the UI action was executed
 2,  12956,         32,          0,  12.17, SCgktqw33c43868414317, 7, SCgktqw33c43868414317, 7
  AC1tko7197f615088600, 1,          LC,    (  WC1n2sgo728683819644,      UIAButton, Recuperar memoria ) [            ]
// TO = state identifier after executing the UI action
 3,  12976,          0,         31,  14.62, SCgktqw33c43868414317, 7, SC13mb0o33c6127651026, 3,
  AC5av6iy7f1081957624, 1,          LC     (   WC836fkw18521259298,      UIAButton,                  5 ) [            ]
// x = number of times each state/action was visited/executed
 4,  12976,          0,         32,  12.80, SC13mb0o33c6127651026, 3, SC13mb0o33c6127651026, 3,
  AC6vkv5z7f1153519684, 1,          LC     (   WClqchm51d961068525,      UIAButton,             Restar ) [            ]
// LC = Left Click
 5,  12976,         46,         15,  17.58, SC13mb0o33c6127651026, 3, SC13mb0o33c6127651026, 3,
  ACq0wpvs7f2433931593, 1,          LC     ( WC1w3r2cr222546437192,      UIAButton,       Multiplicar ) [            ]
...
46,  14552,         15,         93,  29.27, SCg40wonb4e3944825295, 9, SCg40wonb4e3944825295, 9,
  ACnf8i5t7f3638265922, 1,          LC     (  WCpqoaut202987730115, UIARadioButton,               Byte ) [            ]
// WCphm8xo182827665160 is the target widget of the LC action
47,  14552,         47,         47,  22.17, SCg40wonb4e3944825295, 9, SCg40wonb4e3944825295, 9,
   ACu7qc7g7f734729273, 1,          LC     (  WCphm8xo182827665160,      UIAButton,                  0 ) [            ]
// UIAButton is the role of the widget at which the action was performed
48,  14552,         47,         47,  23.33, SCg40wonb4e3944825295, 9, SCg40wonb4e3944825295, 9,
  ACxpmsrr7f2167374728, 1,          LC     (  WCka6ydb282537390550,      UIAButton, Almacenar memoria ) [            ]
// "Borrar memoria" is the TITLE of the widget at which the action was performed
49,  14552,         15,         78,  25.20, SCg40wonb4e3944825295, 9, SCg40wonb4e3944825295, 9,
 AC1oemsk37f2943277776, 1,          LC     ( WC12nbxul253697924011,      UIAButton,    Borrar memoria ) [            ]
// [  ] = no action parameters. Typing actions would display the wrote text between the brackets
50,  14576,         16,         78,  23.92, SCg40wonb4e3944825295, 9, SCg40wonb4e3944825295, 9,
  ACas2sx97f4257138900, 1,          LC     (  WC1cfv19v22135701647, UIARadioButton,            Binario ) [            ]
```

5.4 Tests UI screenshots

You will find at *scrshots/sequence*/*.png* screenshots of the SUT UI for each state visited in the test and screenshots of each executed UI action. The picture file name for states is its identifier and for actions it is in the format: *stateIdentifier_ actionIdentifier.png*.

5.5 Tests graphs

TESTAR represents tests as directed pseudo graphs, which contain loops (an action that makes no reaction in the UI), multiple edges between vertexes (two different actions from the same state make the UI to change to the same state) and multiple vertex targets for edges (the UI could react differently to the same action depending on its internal temporal behaviour).

Graphing data is represented in XML (*graph_ timestamp.xml*) as a set of nodes and links that conform your test, as shown in the commented and simplified sample of listing 8. Nodes represent UI states and links UI actions between states. Test graphs are provided with different perspectives (minimal, tiny, screenshots), which can be found at *graphs/sequence*/**: Graphs are stored by default with .dot[13] extensions. To help in their inspection it is recommended to use the alternative .svg[14] format, which can be visualized through a compatible viewer (e.g. a web browser). Verdicts are also included (PASS as green circle, FAIL as red circle or WARNING in orange like for example a suspicious widget title).

Listing 8: Sample graph XML

```
<?xml version="1.0"?>
<TESTAR_GRAPH version="1.0.20160331"> // a graph consisting of a set of nodes
    (states) and links (actions)
   <graph_states> // set of UI states (as many as test discovered)
// id = concrete identifier, aid = abstract identifier,
// visited = number of times the test visited the state,
// wcount = widgets count in the UI state,
// widgets = the list of widgets and their number of executed actions as pairs of
    identifier=number
// unexecuted = list of derived and unexecuted UI state' actions (abstract
    identifiers)
      <state id="SC1160ylw3862984133231" aid="SR4ygmlp3761868872537" visited="9"
    wcount="45" widgets="{WC13kcegf20254564757=1, WC1rwj1ac1e2906031608=0, ...,
    WCt96dae181894528334=0, WC1c5p1an1f2886007819=0}"
    unexecuted="[AArmtj5u7f655493483,...,AA19wlaph80920180123]"/>
   </graph_states>
   <graph_actions> // set of UI actions (as many as test executed)
// order = actions execution order, id = concrete identifier, aid = abstract
    identifier,
// visited = number of times the test executed the action,
// from = state at which the action was executed, to = the state after the action
    execution
      <action order="1" id="ACaiz3i0804113271233" aid="AAaiz3i0804113271233"
    visited="1" from="SC1160ylw3862984133231" to="SC1160ylw3862984133231"/>
```

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

[14]Scalable Vector Graphics. You might need to convert the graphs from .dot to .svg through the *offline_ graph_ conversion.bat* script, which requires the graphviz tool properly installed (check section ??).

```
      ...
      <action order="25" id="AC1i96cv7e1847397662" aid="AA1i96cv7e1847397662"
    visited="1" from="SCd6aozo389317110099" to="SC1160ylw3862984133231"/>
   </graph_actions>
</TESTAR_GRAPH>
```

Next, the different graphs that can be found are described:

- **Minimal** (graph_timestamp_minimal.*). Figure 10 displays a sample graph. It provides an overview of the test extent, without populating too much information. UI states are illustrated by rectangles with a number, which counts the number of times the test traversed through that particular state. UI actions are shown as links between nodes with a number, which counts for the number of times that particular action was executed in that state. Colors denote a ratio of repetition to help for a direct visual marking of less/more exercised UI. Additionally, pink filled circles and links represent UI actions that were derived in the corresponding state but that were never executed. The number in the pink links account for the number of unexecuted actions. Nodes with thicker outline represent the ones in the longest path.
- **Minimal abstract** (graph_timestamp_minimal_abstract.*). Figure 11 displays a sample graph. As an attempt for the minimal graphs scalability, an abstraction of the graphs is used to clustering sets of related states on the one hand, and sets of related actions on the other hand. Also, the links include the actions order in the test (between brackets).
- **Tiny** (graph_timestamp_tiny.*). Figure 13 displays a sample graph. Tiny version is an enhancement of the minimal version appending states identifiers inside the nodes and actions identifiers[15] inside the links. Moreover, actions order in the test is indicated by the numbers between brackets.
- **Tiny abstract** (graph_timestamp_tiny_abstract.*). Figure 12 displays a sample graph. This is the equivalent abstract tiny version as explained for the minimal versions.
- **Screenshots** (graph_timestamp_scrshoted.*). Figure 14 displays a sample graph. This is the most useful for analyzing your SUT behaviour, enhances previous versions with states and actions screenshots making it ready as test documentation. For the latter, the screenshot applies 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).
- **Screenshots abstract** (graph_timestamp_scrshoted_abstract.*). Figure 15 displays a sample graph. The equivalent abstract version as explained for the minimal versions.
- **Resumed versions**. All previous versions are also available for resumed graphs, those which are the product of resuming from a previous test (concretely, from a previous XML graph). The corresponding[16] test resumed versions samples are displayed at figures 16, 17, 19, 18, 20 and 21. Three styles for nodes' outline are applied to indicate:

[15] An identifier that starts with G_* denotes a group of similar actions (e.g. two different typing actions at the same widget) to prevent massification of links

[16] Screenshots are only displayed for the last test sequence

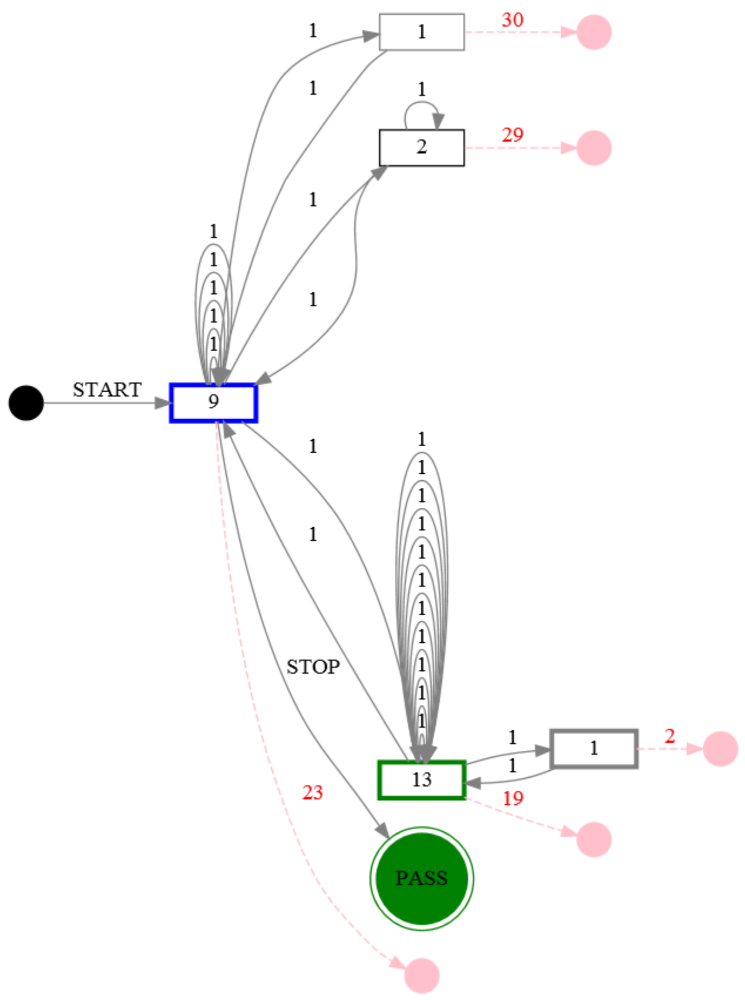


Figure 10: Minimal graph

a) dotted: states traversed by a previous test, b) corners diagonals: a) states which have been revisited by the test and c) straight: new discovered UI states. Similarly, links also apply these outlines with the exception of case b), for which a tapered style is used instead. Additionally, thicker outlined links indicate multi-target actions, those that reach different states due to varying SUT behaviour.

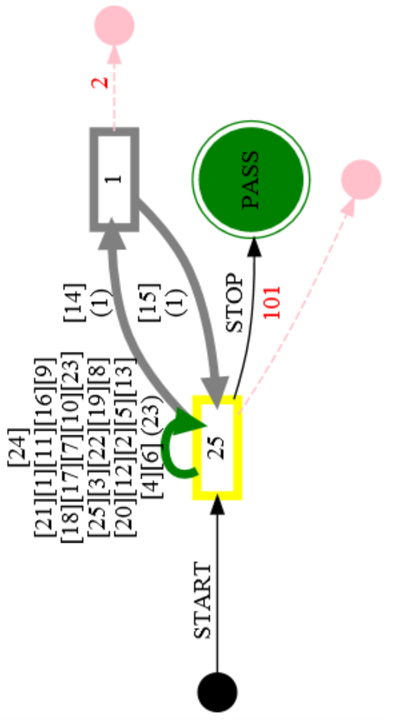


Figure 11: Minimal abstract graph

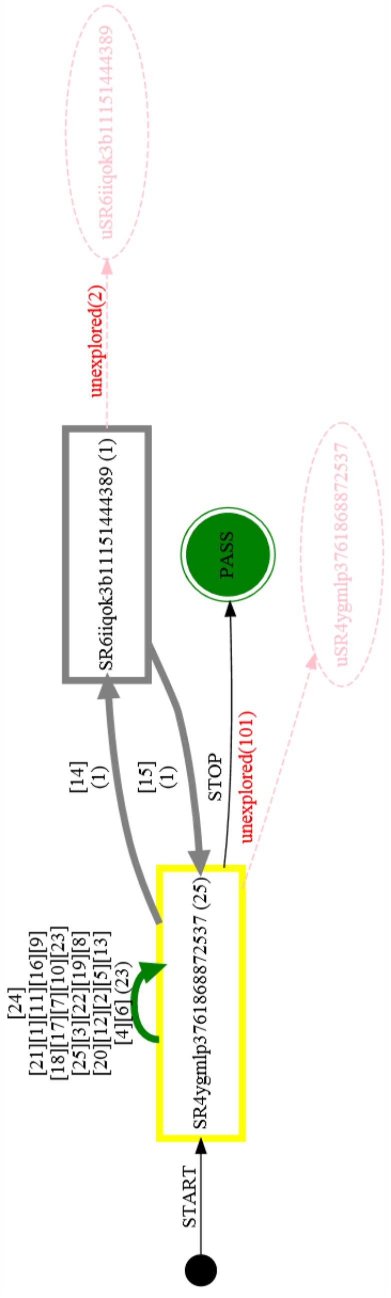


Figure 12: Tiny abstract graph

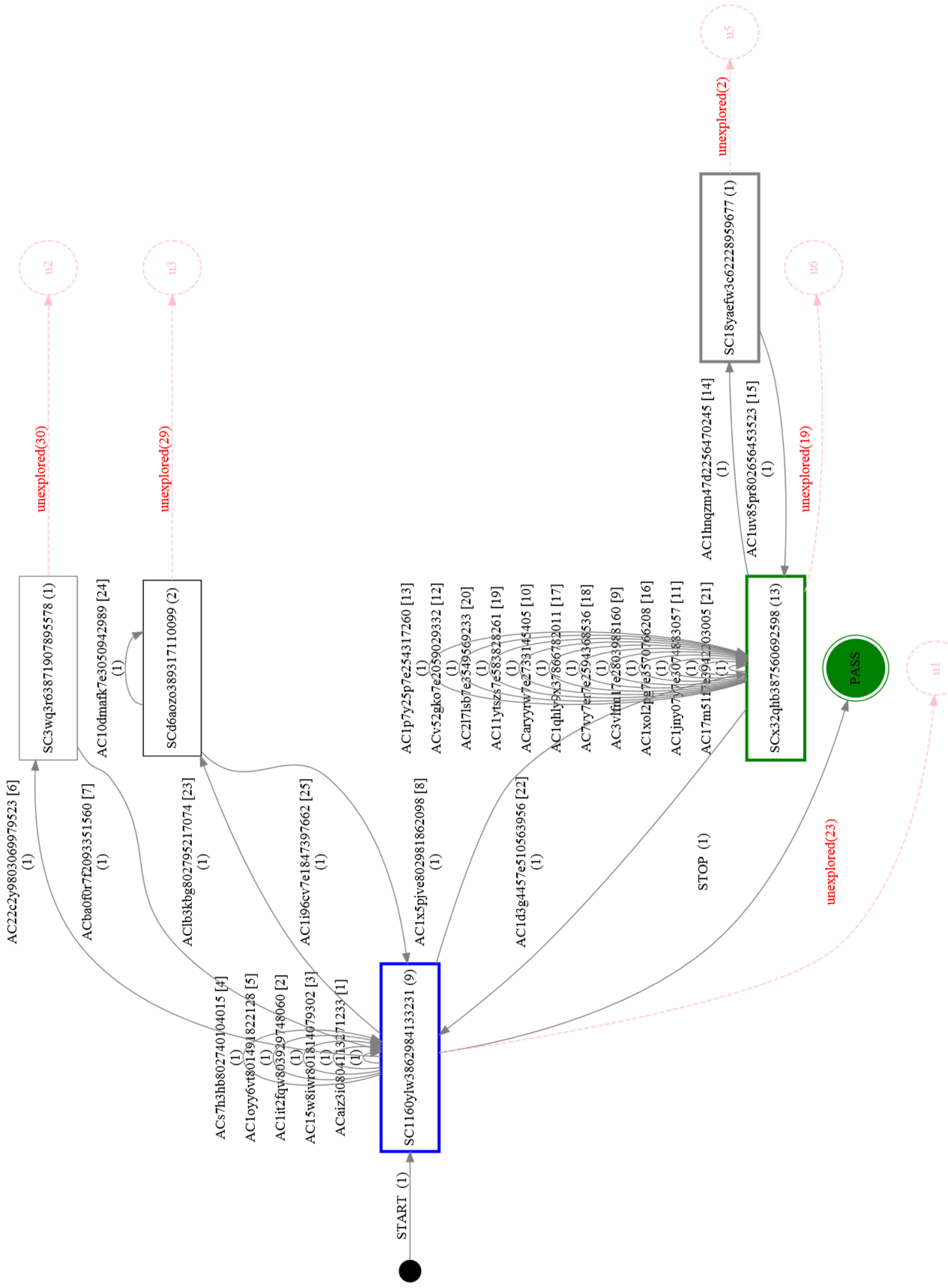

START (1)
SC1160ylw3862984133231 (9)
ACs7h3hb802740104015 [4] (1)
AC1oyy6vt801491822128 [5] (1)
AC1it2fqw803929748060 [2] (1)
AC15w8iwr801814079302 [3] (1)
ACaiz3i080411327123 [1] (1)
AC22c2y9803069979523 [6] (1)
ACba0f0r7f2093351560 [7] (1)
AClb3kbg802795217074 [23] (1)
AC1i96cv7e1847397662 [25] (1)
AC1x5pjve802981862098 [8] (1)
AC1d3g4457e510563956 [22] (1)
STOP (1)
unexplored(23)
SC3wq3r63871907895578 (1)
unexplored(30)
u2
AC10dmafk7e3050942989 [24] (1)
SCd6aozo38931711009 (2)
unexplored(29)
u3
AC1p7y25p7e254317260 [13] (1)
ACv52gko7e2059029332 [12] (1)
AC2l7lsb7e3549569233 [20] (1)
AC11ytszs7e583828261 [19] (1)
ACaryyrw7e2733145405 [10] (1)
AC1qhly9x378667832011 [17] (1)
AC7vy7er7e2594368536 [18] (1)
AC3vlfm17e2803988160 [9] (1)
AC1xol2pg7e3570766208 [16] (1)
AC1jny07i7e3074883057 [11] (1)
AC17m517e3942203005 [21] (1)
SCx32qhb387560692598 (13)
AC1hnqzm47d2256470245 [14] (1)
AC1uv85pr802656453523 [15] (1)
SC18yaefw3c62228959677 (1)
unexplored(2)
u5
unexplored(19)
u6
PASS
u1

Figure 13: Tiny graph

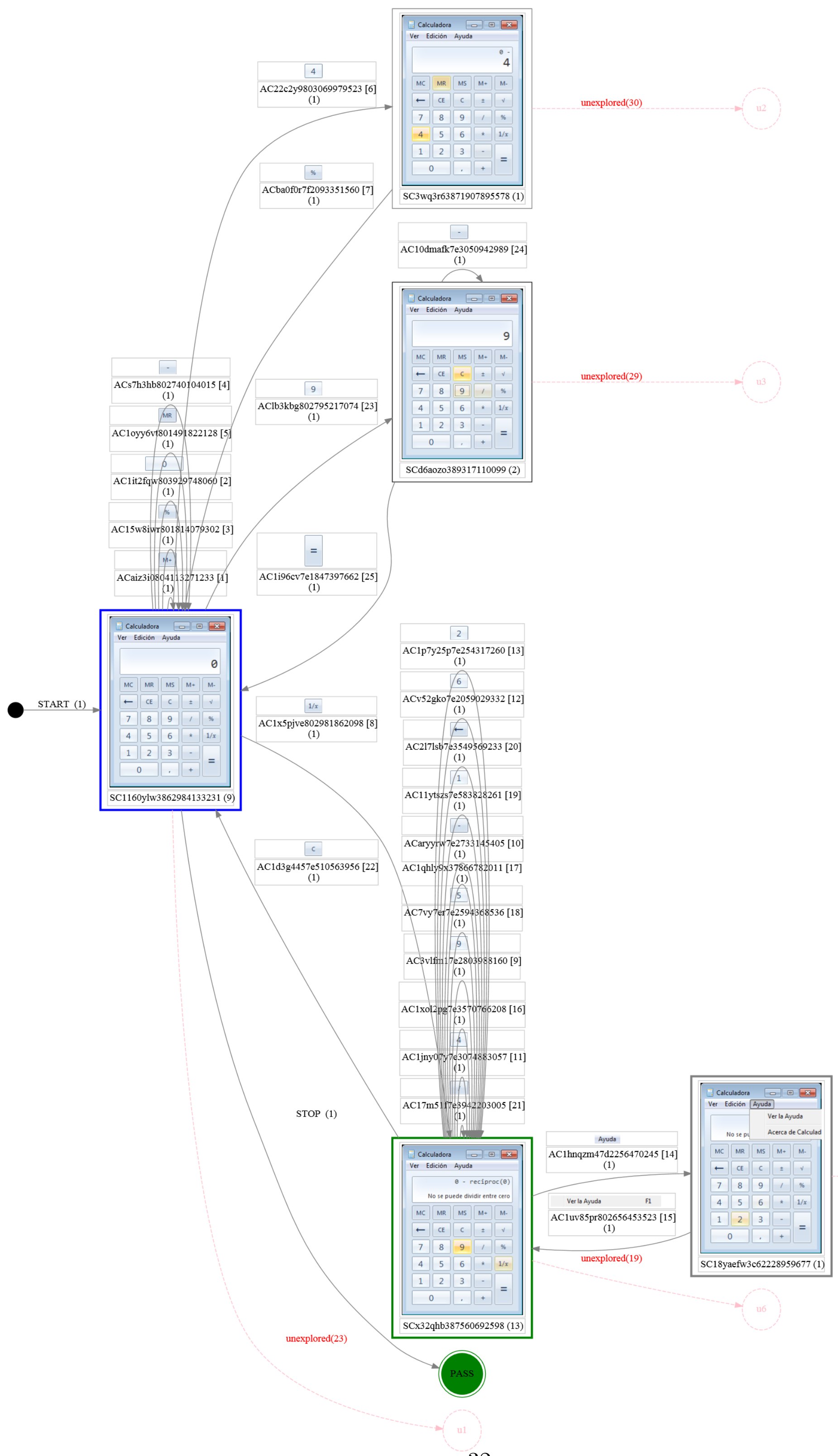

AC22c2y9803069979523 [6]
(1)
ACba0f0r7f2093351560 [7]
(1)
SC3wq3r63871907895578 (1)
unexplored(30)
u2
AC10dmafk7e3050942989 [24]
(1)
ACs7h3hb802740104015 [4]
(1)
AC1oyy6vt801491822128 [5]
(1)
AC1it2fqw803929748060 [2]
(1)
AC15w8iwr801814079302 [3]
(1)
ACaiz3i0804113271233 [1]
(1)
AClb3kbg802795217074 [23]
(1)
SCd6aozo389317110099 (2)
unexplored(29)
u3
AC1i96cv7e1847397662 [25]
(1)
START (1)
SC1160ylw3862984133231 (9)
AC1x5pjve802981862098 [8]
(1)
AC1d3g4457e510563956 [22]
(1)
AC1p7y25p7e254317260 [13]
(1)
ACv52gko7e2059029332 [12]
(1)
AC2l7lsb7e3549569233 [20]
(1)
AC11ytszs7e583828261 [19]
(1)
ACaryyrw7e2733145405 [10]
(1)
AC1qhly8x37866782011 [17]
(1)
AC7vy7er7e2594368536 [18]
(1)
AC3vlfm17e2803988160 [9]
(1)
AC1xol2pg7e3570766208 [16]
(1)
AC1jny07y7e3074883057 [11]
(1)
AC17m51f7e3942203005 [21]
(1)
STOP (1)
AC1hnqzm47d2256470245 [14]
(1)
AC1uv85pr802656453523 [15]
(1)
SC18yaefw3c62228959677 (1)
unexplored(2)
u5
unexplored(19)
u6
SCx32qhb387560692598 (13)
unexplored(23)
PASS
u1

Figure 14: Screenshots graph

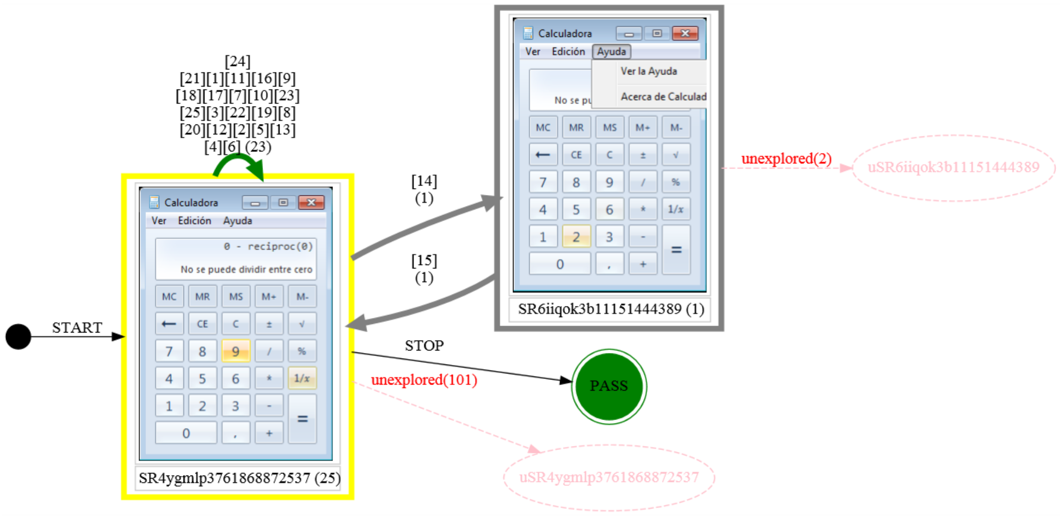


Figure 15: Screenshots abstract graph

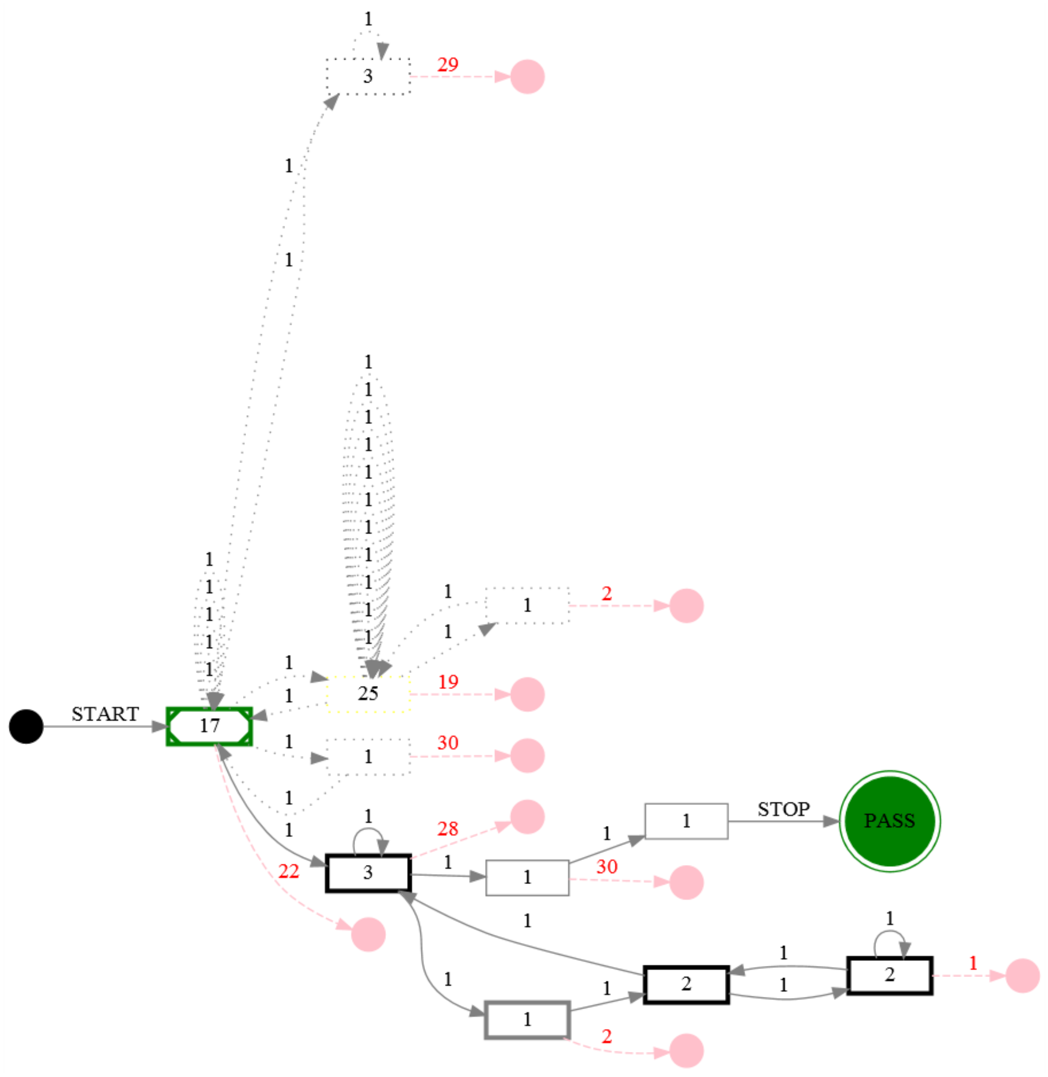


Figure 16: Resumed minimal graph

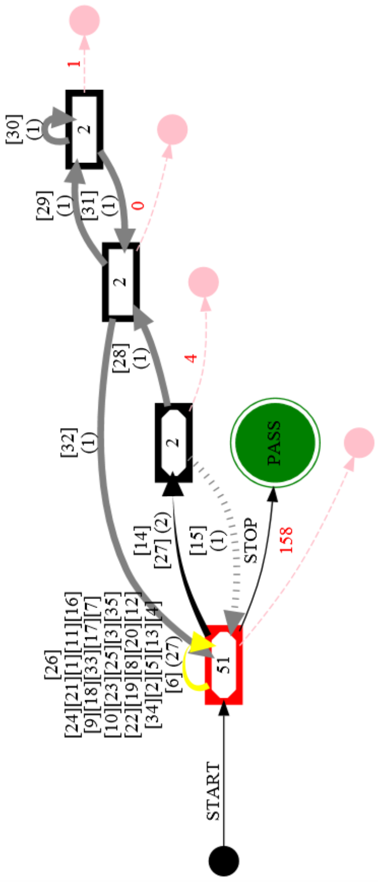


Figure 17: Resumed minimal abstract graph

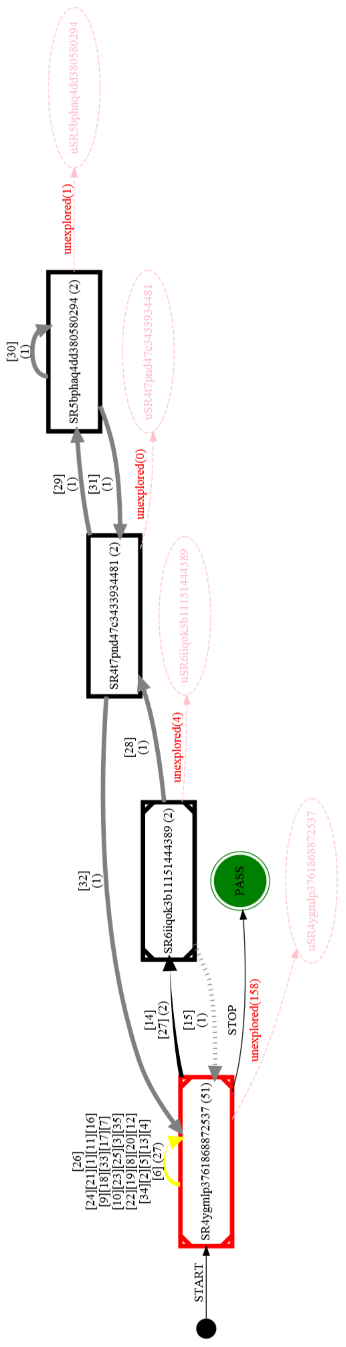


Figure 18: Resumed tiny abstract graph

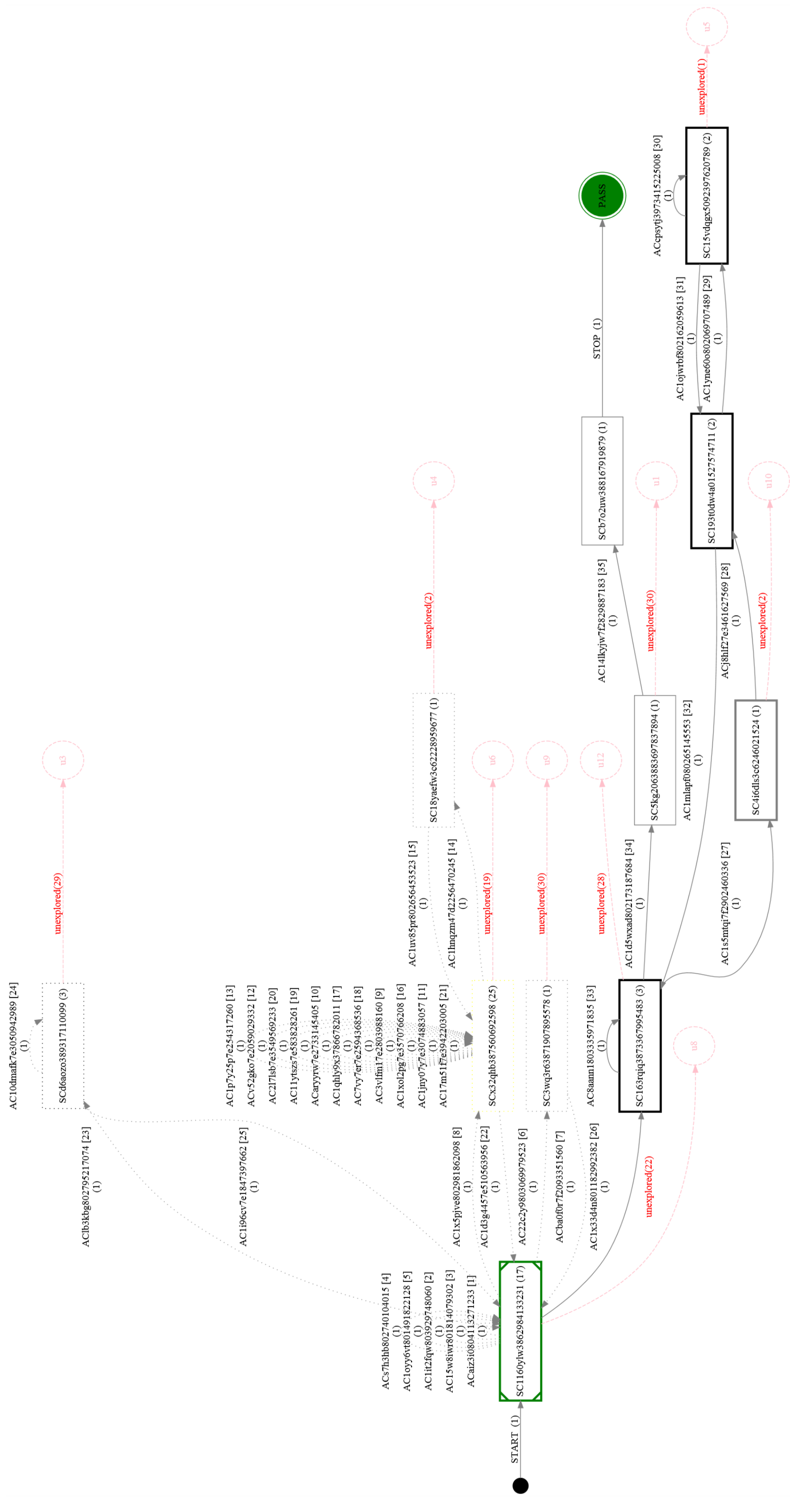

START (1)
SC1160ylw3862984133231 (17)
SCx32qhb387560692598 (25)
SC3wqxr63871907895578 (1)
SC163rqiq387336799548 3 (3)
SCd6aozo389317110099 (3)
SC18yaefw3e622289596 77 (1)
SC5kg20638836978378 94 (1)
SC4i6dls3e6246021524 (1)
SCb7o2uw38816791987 9 (1)
SC193i0dw4a0152757471 1 (2)
SC15vdqgx50923976207 89 (2)
PASS
STOP (1)
unexplored(29)
unexplored(2)
unexplored(19)
unexplored(30)
unexplored(28)
unexplored(22)
unexplored(1)
u3
u4
u6
u9
u12
u1
u10
u8
u5

Figure 19: Resumed tiny graph

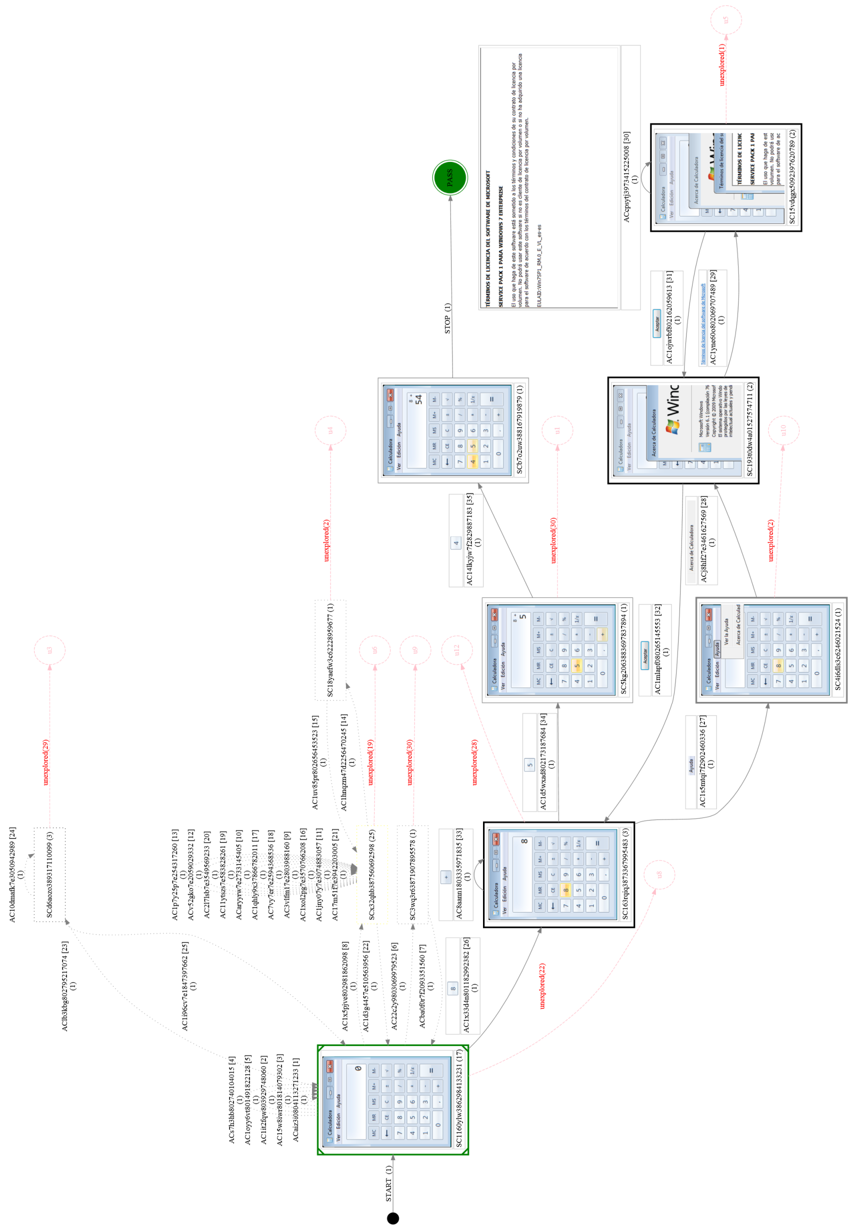


Figure 20: Resumed screenshots graph

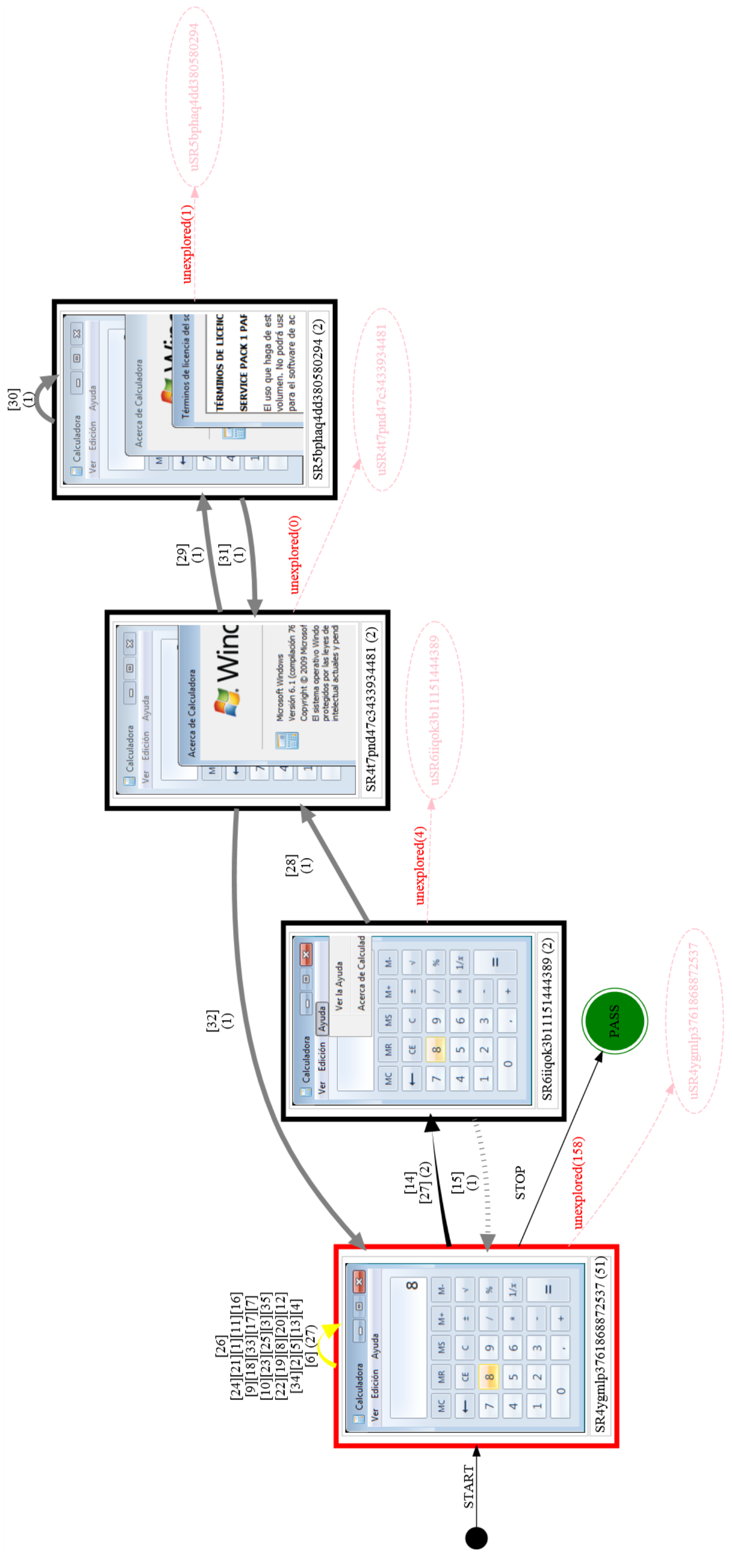


Figure 21: Resumed screenshots abstract graph

5.6 Tests metrics

Indicators about a test performance are provided through three sources of informaiton:

- **CSV metrics** (output/metrics/*.csv). An example is provided in tables 2, 3 and 4. Next, it is detailed how to interpret these metrics:
  - Verdict: did the test PASS or FAIL?
  - FAILS: if *ForceToSequenceLength* property was set to true (check section D) it displays the number of times the SUT failed until the sequence length is reached.
  - minCvg/maxCvg: for all the UI states visited by the test, displays the minimum/maximum UI actions[17] coverage[18] achieved.
  - maxpath: the test' longest executed UI path[19], as the number of graph states.
  - graph-states: the concrete UI states number visited by the test.
  - abstract-states: the abstract UI states number visited by the test.
  - graph-actions: the sum of all derived[20] UI actions for all the UI states visited by the test.
  - test-actions: the number of test executed UI actions.
  - SUTRAM: the peak SUT used RAM during the last test sequence.
  - SUTCPU: the peak SUT used CPU during the last test sequence.
  - TestRAM: the peak RAM used by TESTAR during the last test sequence.
  - TestCPU: the peak[21] delay between UI actions during the last test sequence.
  - fitness: a combination of other metrics to roughly estimate how good[22] the last test sequence was.

Verdict	**FAILS**	**minCvg(%)**	**maxCvg(%)**	**maxpath**
PASS	0	3.23	33.33	8

Table 2: Example CSV metrics 1/3

graph-states	**abstract-states**	**graph-actions**	**test-actions**
13	2	24	25

Table 3: Example CSV metrics 2/3

- **STDOUT metrics** (output/*.dbg.log). More indicators can be found at STDOUT (or *.dbg.log files if *powershell* was accessible). A simplified and commented sample is displayed in listing 9.

[17]As specified by the test set up
[18]An UI action was covered if the test did execute it almost once
[19]How far in the UI did the test go
[20]As specified by the test set up
[21]Note: the first UI action will at minimum take as long as the SUT startup time
[22]0.0 is best, 1.0 is worst

SUTRAM(KB)	SUTCPU(%)	TestRAM(MB)	TestCPU(s)	fitness
220.0	51.06	80.5	0.247	6.500778E-4

Table 4: Example CSV metrics 3/3

Listing 9: Sample STDOUT metrics

```
// live test metrics at STDOUT (one line per executed UI action)
S[1=1]-A[  1] <  0@  0 KCVG>... SR =    12640 KB / SC =   0.00% ... TC: 1.927
    s / TR: 123.0 MB ... L/S/T: 3/2/0
S[1=1]-A[  2] <  3@ 31 KCVG>... SR =    12772 KB / SC =   0.00% ... TC: 0.205
    s / TR: 123.0 MB ... L/S/T: 1/4/1
...
S[1=1] // test sequence 1 (left), saved as sequence number 1 (right)
-A[ 50] // UI action number 50 in the test order
< 45@10a KCVG> // 45% coverage (left) of the known UI (right) as the number
    10a (a = x10, b = x100, c= x1000, etc.) of derived UI actions
... SR =    18036 KB / SC =   0.00% // SR = SUT RAM, SC = SUT CPU
... TC: 0.170 s / TR: 145.5 MB // TC = TESTAR CPU (delay between UI actions),
    TR = TESTAR RAM usage (some test set ups will consume higher memory; e.g.
    graphing of long tests with non random algorithms for action selection)
L/S/T: 1/2/3 // serialization queues (Logs, Screenshots, Test java objects)
...
S[1=1]-A[100] < 60@16a KCVG>... SR =    19480 KB / SC =  57.14% ... TC: 0.065
    s / TR: 126.0 MB ... L/S/T: 5/5/0

// final metrics at test sequence end
verdict,FAILS,minCvg(%),maxCvg(%),maxpath,graph-states,abstract-states,
   PASS,    0,     0.00,    99.90,      4,          10,               9,
graph-actions,test-actions,SUTRAM(KB),SUTCPU(%),TestRAM(MB),TestCPU(s),
          100,         100,    6840.0,   114.29,      145.5,     0.774
fitness = 1.1828199545645511E-4 // 0.0 best, 1.0 worst (based on all metrics)
```

- **Tests logs metrics** (output/logs/*.log). Check section 5.3.

5.7 How to analyse tests data

Lets assume TESTAR automated tests discovered issues in your SUT (e.g. unexpected exceptions). You will know if you see folders like *output/sequences_warning*, *output/sequences_suspicioustitle*, *output/sequences_unresponsive*, *output/sequences_unexpectedclose*, *output/sequencces_fail*, or *output/sequences_other*. You can also notice it by a verdict metric different than *PASS*.

So, we have a faulty test sequence. Then, how do we analyse the cause to help on defect correction? Next, it is provided the typical process flow to analyse the reported tests data:

1. Check the screenshots version of the graphs, corresponding to the faulty test sequences. In the graph, use the viewer (e.g. web browser) text finding utility and search for the word *FAIL*. Or alternatively, check the last[23] UI action that was executed in the test

[23]Search the text *[action_order]*, which you can get from the logs and/or metrics

sequence. The goal is to move to the test ending, inspecting the last UI states and actions because all of the previous test actions were executed successfully with the verification oracles that you did specify in your test set up. In most cases, the last UI state and action might reveal the cause of the fault. Use other graph versions and states/actions identifiers to retrieve as much information as you need in your analysis.

2. If the graphs did not reveal useful hints about the fault (e.g. missing screenshots), then you can still check the detailed ordered executed actions logs. Open the corresponding actions table log as described in section 5.3 (logs/*_testable.log). Jump to the end of the log (last executed actions) and revise in reverse ordering the executed actions and traversed UI states. You can use states and actions identifiers to know about them (screenshots, detailed[24] actions information). The goal is to figure out the UI context and actions at which the fault did emerge.

3. Yet, you might still not be able to deduce the cause of the fault from graphs and logs (e.g. the information about states and actions is not complete or not detailed enough). Several open approaches follow: a) view working mode (section 3.4), which most probably will not provide any additional hint (but you might want to give it a chance) b) replay working mode (section 3.3), where you might come into a non reproducible test, c) preparing your test set up for further refined tests oriented towards the UI part that revealed the fault or d) checking your SUT logs, if any.

4. Did you reach this point? You know about a fault in your SUT but you were unable to understand the cause? Then, prepare your SUT test environment to provide clues (the missing information for your analyses). For example, make sure that no crucial SUT debugging information will be missing after a test execution.

5. At a final step, once you succeed to identify the UI context (states and actions) for the fault, you can prepare your test set up to perform stress testing over that specific UI context. Just start with a test protocol that derives no actions at all, and enable these actions that are relevant for the fault.

[24]Listing 3

6 Further documentation

For more information, check the next available online resources:

- FAQ (Frequently Asked Questions): https://testar.org/faq/
- Tool website: https://www.testar.org, for up to date information
- *GitHub* repository: https://github.com/STaQ-PROS-UPV/TESTAR, for technical information. You might also want to submit pull requests in the project: https://github.com/STaQ-PROS-UPV/TESTAR/pulls

7 Experimental features

TESTAR development is ongoing. Some unfinished tasks of which you might find signs and hints on the code are listed next:

- AdhocTest mode: it allows sending commands (UI actions) to TESTAR through a socket (port 47357). For example, sending a message with the format

  ```
  <action_type(parameters*)>\r\n // e.g. LC(500,420)
  ```

  would indicate to TESTAR that it should left click (LC) at absolute screen position (500,420). This feature is unused and deprecated.
- Evolutionary algorithm for action selection: it is part of an ongoing research and is thus experimental.
- Prolog based algorithms for action selection: it is part of an ongoing research and is thus experimental.
- Semi-automated specification of oracles: it is part of an ongoing research and is thus experimental.

8 Known issues

Known issues are enumerated next:

1. Graphical environment: color = 32bit is required. Using remote desktop with lower color (i.e. 16/24 bit) will result on an **"Unable to update layered window"** error message (@console)

2. System Under Test: browsers support for Web applications:

   a) *Internet explorer x86* (32 bit executable) will result on a **"System is offline! I assume it crashed"** error message (@console).

   b) *Internet explorer x64* (64 bit executable) might result in undetected UI widgets. IE automation might stop working because of *oleacc.dll*, which is a windows system library responsible for the Active Accessibility support. To fix, you will need to register the library in windows registry: open an elevated command line (cmd.exe with admin rights) and execute the command "regsvr32.exe oleacc.dll" IE automation might also fail because of security settings. On Windows server is very likely that IE has default security settings that prevents scripting. You have to allow active scripting for the site you want to automate: "IE Tools/Internet Options/Security tab/Custom level button/Scripting/Allow active scripting".

   c) *Mozilla Firefox*: scrollbars not detected (not compliant with Accessiblity API).

   d) *Google Chrome*: web application widgets not being detected.

3. Protocol editor: **"ReferenceError: ImportPackage is not defined in <eval> at line number 2"** error[25] message (just ignore).

4. Protocol compilation: **"JDK required (running inside of JRE)"** error message. Make sure you are running TESTAR with Java JDK version (not JRE version). Did you double click *testar.jar*? Check the *run.bat* script to know about how it should be run for a JDK JVM (x64).

5. Console error messages:

   a) "System is offline! I assume it crashed": check whether the SUT is already running (close or finish its process as required) Yet, TESTAR will automatically try to detect and kill any SUT running processes.

   b) OutOfMemory exceptions: TESTAR might consume too much memory in long tests. To solve, use random algorithm, disable graphs and/or edit the *run.bat* script and modify the line "set MEM=n" with a value for n in number of Gigabytes, which establishes the JVM memory allocation pool. You can check the used TESTAR RAM from reported metrics (see section 5).

[25]Due to outdated library: jsyntaxpane-0.9.5-b29

A Widgets and their properties

A.1 UI widget

A **Widget** is a graphical element in the UI of the System Under Test (SUT). General examples are: windows, titlebars, scrollbars, toolbars, menubars, statusbars, progress-bars, trees, buttons, radio buttons, checkboxes, comboboxes, lists, list items, tables, data items, labels, texts, text boxes, tooltips, panels, groups, spin-boxes, sliders, separators, rulers, headers, footers, menu items, popup menus, password fields, page tabs, tab items, icons, images, canvas, hyperlinks, etc. There can also be **customised widgets** with unknown visual appearances and functionalities, which are specific to the SUT.

The more general definition of a widget would be a placeholder with a graphical aspect at which the user could perform an interaction like pressing, selecting, inspecting a piece of information, getting progress feedback of an activity (e.g. loading a file), etc. **Widgets conform the API for the user to interact with the SUT.** A widget is usually characterized by a particular behaviour. For example, the user could click a button. It is also characterized by a set of properties that define its visual aspect in the screen (e.g. position, shape, color, text).

A.2 Accessibility technology

We use the Accessibility API – which simplifies computer usage for people with disabilities – to obtain the SUT's GUI state. It allows to gather information about the visible widgets of an application and gives TESTAR the means to query their property values. After querying the application's GUI state, we save the obtained information in a so-called *widget tree* which captures the structure of the GUI.

Figure 22 displays an example of such a tree. Each node corresponds to a visible widget and contains information about its type, position, size, title and indicates whether it is enabled, etc. The Accessibility API gives access to over 160 properties which allows us to retrieve detailed information such as:

- The **type** of a widget.
- The **position** and **size** which describe a widget's rectangle (necessary for clicks and other mouse gestures).
- It tells us whether a widget is **enabled** (It might not make sense to click disabled widgets).
- Whether a widget is **blocked**. This property is not provided by the API but we calculate it. Figure **??** shows a message box which blocks all other widgets behind it. Our monkey detects those and other modal dialogs (like menus) and sets the blocked attribute accordingly.
- Whether a widget is **focused** (has keyboard focus) so that the monkey knows when it can type into text fields.
- Attributes such as **title, help** and other descriptive attributes are very important to distinguish widgets from each other and give them an identity. We will make use of this in the next subsection when we describe our algorithm for action selection.

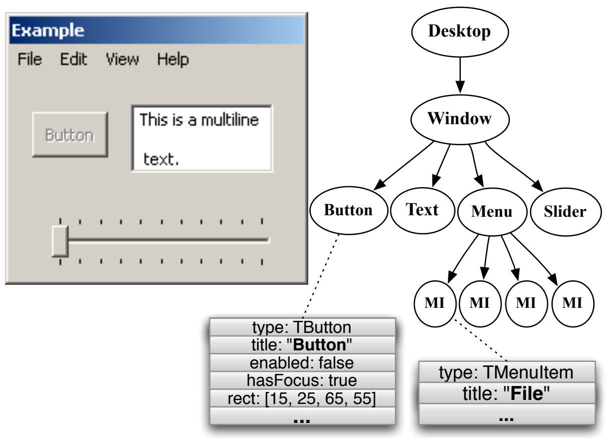


Figure 22: The state of a GUI can be described as a widget tree which captures property values for each control.

This gives us access to almost all widgets of an application, if they are not custom-coded or drawn onto the window. We found that the Accessibility API works very well with the majority of native applications (since the API works for all standard widgets, the developers do not have to explicitly code for it to work).

A.3 Widget properties

As indicated, the Accessibility API gives access to over 160 properties of the widgets. The properties of each widgets characterises the visual appearance and behaviour of that particular widget. In this section we list the most[26] useful properties for test automation in TESTAR:

- **Role**: determines the native widget type (e.g. UIAButton in Windows platforms) and, implicitly, its expected behaviour (e.g. clicking a button or typing in a text box).
- **ControlType**: the native widget type as a code (check *Role* property).
- **ClassName**: the widget type (check *Role* property). For example: *Button.*
- **AutomationId**: an identifier (could be empty) to uniquely represent the widget.
- **Title**: the text (might be empty) displayed in the widget (e.g. "Close" in a button that closes a window).
- **Name**: the name of the widget (could be empty).
- **Shape**: screen position (x, y) and dimensions (width, height).
- **Enabled**: true or false. Indicates whether the widget will react to user inputs (e.g. a left click).

[26]Other properties can be easily extended to TESTAR from the accessibility technologies.

- **ZIndex**: the Z-ordering of the widget in the User Interface, which determines whether it is in foreground (higher value) or in background (lower value).
- **Orientation**: widget orientation as none (0), horizontal (1) or vertical (2).
- **ScrollPattern**: true or false. Whether a scrolling pattern is available.
- **HorizontallyScrollable**: true or false. Whether a widget is an horizontal scroller.
- **ScrollHorizontalPercent**: for an horizontal scrollable widget, determines the amount (%) of horizontal scrolling applied.
- **ScrollHorizontalViewSize**: for an horizontal scrollable widget, determines the horizontal scrolling window used.
- **VerticallyScrollable**: true or false. Whether a widget is a vertical scroller.
- **ScrollVerticalPercent**: for a vertical scrollable widget, determines the amount (%) of vertical scrolling applied.
- **ScrollVerticalViewSize**: for a vertical scrollable widget, determines the vertical scrolling window used.
- **Desc**: a descriptive text of the widget.
- **ProviderDescription**: technical descriptive text of the widget.
- **ToopTipText**: a floating descriptive text displayed to the user (could be empty).
- **HelpText**: a help text displayed to the user (could be empty).
- **IsKeyboardFocusable**: true/false. Whether the widget can gain keyboard focus.
- **HasKeyboardFocus**: true/false. Whether the keyboard focus is at the corresponding widget.
- **WindowVisualState**: for widgets of type *Window* determines its visual state as normal (0), maximized (1) or minimized (2).
- **WindowInteractionState**: for widgets of type *Window* determines the interaction state as running (0), closing (1), ready for user iteraction (2), blocked by modal window (3) and not responding (4).
- **FrameworkId**: a string containing the name of the underlying UI framework that the automation element belongs to. It enables to process automation elements differently depending on the particular UI framework (e.g. "Win32", "WinForm", and "DirectUI" for Windows platforms).

Additionally, TESTAR provides derived (calculated) properties:

- **Path**: a sequence of numbers that determine the widget location in the widgets-tree hierarchy.
- **Ancestors**: a list of *Roles* that determine the parent widgets in the widgets-tree hierarchy.

- **ConcreteID**: an identifier that uniquely represents the widget (see section C for a more detailled explaination about identifiers).
- **Abs(R)ID**: an identifier that abstractly represents the widget by its *Role* property. Again, we refer to section C for a more detailled explaination about identifiers.
- **Abs(R,T)ID**: an identifier that abstractly represents the widget by its *Role* and *Title* properties.
- **Abs(R,T,P)ID**: an identifier that abstractly represents the widget by its *Role*, *Title* and *Path* properties.
- **State**: an identifier that concretely represents the widget-tree.
- **Blocked**: true or false. Whether the widget is blocked by other widgets in the UI (e.g. other widget is on top/foreground or the widget is not part of a modal window). A blocked widget should not react to user inputs (but a Fault could be discovered doing so).
- **IsWindowModal**: true or false. Whether the widget behaves as a modal window (any widget outside its tree hierarchy will be *Blocked*).

Note that not all the properties may be available for every widget, and even if the property is available it may have no value.

B UI States and Actions

B.1 UI States

The User Interface (UI) of a software application is composed of all the possible graphical elements that could be displayed to the end-user. It includes all the windows, dimensions, positions on screen, widgets shapes and visual aspects, data populated in tables, displayed texts, and so on.

Consequently, we define the **UI State** of a SUT, as a concrete widget-tree at a particular timestamp that captures the structure of the UI at that particular moment. The hierarchy in the tree represents parent/child relationships between widgets like *a menu bar* is part of a *window*, *menu items* are part of a *menu* or *list items* are part of a *list*. The root of the three represents the SUT UI and embrace as children all its pertaining windows at a particular timestamp.

A **Widget-tree** wt is defined as $wt = \{root, subtree_1, \ldots, subtree_n\}$ where each of the n subtrees are a child widget-tree of the *root* widget, leaves are UI widgets like buttons, labels, list items and parent widgets in the tree are UI containers like windows, menus, lists, tables, and so on. Figure 23 displays a simplified sample widget-tree for the *Calculator* SUT. Each box corresponds to a widget displayed on screen, for which the sample only shows the properties *Role* and *Title*. In a realistic scenario, and more complex software UI, the widget-tree can easily have hundreds of widgets.

B.2 UI Action

We define an **Action** as a user event (interaction) on the UI like clicking buttons and menu items, typing text in text boxes or dragging scrollbars. Thus, all actions are performed in a particular UI state, so every action belongs to a state. Particularly, some actions might belong to specific widgets in the widget-tree (e.g. a left click is performed in a widget, but pressing the *ESC* key is not). Consequently, for each UI state s there is a set of feasible UI actions $A(s)$, specified in the test protocol (check section 4), as $A(s) = \{a_1, \ldots, a_n\}$. Each action a is defined by a set of properties as $a = \{p_1, \ldots, p_k\}$. For example, {{Role, LeftClick}, {Target, {168.5, 71.5}}} represents a left clicking action at screen position x = 168.5 and y = 71.5.

Finally, we define a **State transition** $t(a)$ as the result of executing an action a as $t(a) = s \rightarrow s'$ where s is the state at which the action was performed and s' the state after its execution (they could be equal or different). Thus, a TESTAR test sequence is defined as $test = s_0 \rightarrow_{a_1} s_1 \rightarrow_{a_2} s_2 \rightarrow \cdots \rightarrow_{a_t} s_t$, where s_0 is the UI state once the SUT is started and s_t the final state after the test sequence execution. For each step in the test sequence TESTAR chooses an action from the set of feasible actions $A(s_i)$.

A list of supported UI actions is enumerated next, yet this list can be easily extended:

- Mouse-move: moves the mouse pointer to a new location (e.g. a widget position).
- Left-click: presses left mouse button at current mouse pointer location.
- Right-click: presses right mouse button at current mouse pointer location.
- Left-double-click: presses twice the left mouse button.
- Left-triple-click: presses three consecutive times the left mouse button.

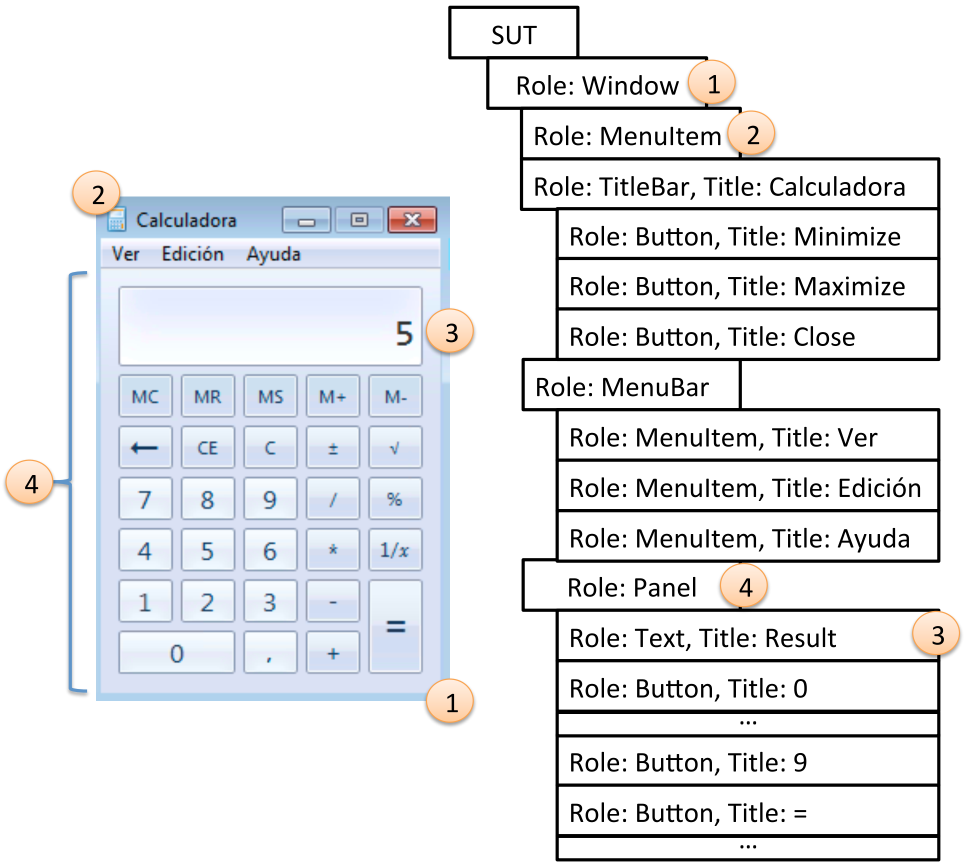

Calculadora
Ver Edición Ayuda
5
MC MR MS M+ M-
← CE C ± √
7 8 9 / %
4 5 6 * 1/x
1 2 3 - =
0 , +
SUT
Role: Window 1
Role: MenuItem 2
Role: TitleBar, Title: Calculadora
Role: Button, Title: Minimize
Role: Button, Title: Maximize
Role: Button, Title: Close
Role: MenuBar
Role: MenuItem, Title: Ver
Role: MenuItem, Title: Edición
Role: MenuItem, Title: Ayuda
Role: Panel 4
Role: Text, Title: Result 3
Role: Button, Title: 0
...
Role: Button, Title: 9
Role: Button, Title: =
...

Figure 23: Sample widget-tree representing the UI state at a concrete timestamp.

- Drop-down (left-click + right-arrow keys): a left click followed by the right arrow key press. It may expand some menus on some SUTs.
- Drag-from-to: drags from one location to another. For example, dragging an object from a palette to a drawing canvas.
- Click-type-into: clicks on a screen location and writes some text on keyboard focus.
- Hit-key: presses a keyboard key.
- Slide-from-to (TESTAR automated): commonly used for sliders.
- Kill-process (TESTAR automated): kills unwanted processes that may be started by actions performed during a test sequence.
- Activate-system (TESTAR automated): brings the SUT window to foreground (in top of any other windows on screen).

C Identifiers for Widgets, States and Actions

TESTAR uses identifiers for its action selection algorithms and reporting. An identifier helps to uniquely reference each widget in the screen. Consequently, an identifier can be calculated for a UI state. Additionally, UI actions are also referenced by an identifier. Next, the formulas applied to calculate the identifiers for widgets, states and actions are presented:

- **ID formula**: TESTAR applies a formula f that converts, with low collision, a text t of varying length into a shorter representation as $f(t) = \mathit{hashcode}(t)+\mathit{length}(t)+\mathit{crc}32(t)$ where + is the concatenation of strings.
- **Concrete Widget ID**: the ID for a widget w is calculated as $\mathit{id}(w) = f(\mathit{Role} + \mathit{Title} + \mathit{Enabled} + \mathit{Path})$ where + is the concatenation of strings from the values of the corresponding properties. An example text for an enabled “ok” button would be: Buttonoktrue0,0,1 (“0,0,1" being the path in the widget-tree). Then, f is applied to the text.
- **Abstract Widget ID**: similar to the concrete widget ID, but using a particular set of properties like $\mathit{Abs}(R)$ for *Role*, $\mathit{Abs}(R,T)$ for *Role* and *Title* and $\mathit{Abs}(R,T,P)$ for *Role*, *Title* and *Path*.
- **Concrete State ID**: the ID for a state s is calculated as $\mathit{id}(s) = f(\mathit{id}(w)_1 + \cdots + \mathit{id}(w)_n)$ where the order of the widgets in the tree hierarchy is preserved in the calculation.
- **Abstract State ID**: similar to the concrete state ID, but using the abstract widgets ID.
- **Concrete Action ID**: the ID for an action a in state s is calculated as $\mathit{id}(a) = f(\mathit{id}(s) + p_1 + \cdots + p_k)$ where p_i are the action properties.
- **Abstract Action ID**: similar to the concrete action ID, but discarding properties of the action (e.g. texts typed, keys pressed). For example, we can refer to writing in a concrete textbox, no matter of the text used to write.

Additionally, a prefix is used for the identifiers to distinguish between:

- Concrete widgets: "WC" prefix (e.g. WC1ef69q9183894040580).
- Abstract (Role) widgets: "WR" prefix (e.g. WR1h6ibpb92248851252).
- Abstract (Role,Title) widgets: "WT" prefix (e.g. WTew6teaa2733511101).
- Abstract (Role,Title,Path) widgets: "WP" prefix (e.g. WP54gyib14378315832).
- Concrete states: "SC" prefix (e.g. SCgktqw33c43868414317).
- Abstract (Role) states: "SR" prefix (e.g. SR1kx62t53b13002840823).
- Concrete actions: "AC" prefix (e.g. AC1hplft17f3457170962).
- Abstract actions: "AA" prefix (e.g. AA1hplft17f3457170962).

Consequently, the equality of widgets, states and actions depend on the equality of identifiers. Abstraction provides the capability to cluster widgets, states and actions, and so the graphs (check sections 5.3 and 5.5).

D Test settings

The first step to test your software application is to set up your test settings. We provide a predefined set of settings for desktop and web applications. You can find them under the *settings* folder at the tool installation folder. Each test settings configuration is stored inside a unique subfolder (e.g. *desktop_generic*), which contains: a) a Java source file (e.g. *Protocol_desktop_generic.java*) with the executable test protocol, described in section 4 and b) a *test.settings* file, which contains a list[27] of test properties[28] described next:

- **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[29] (e.g. .*[cC]lose.*|.*[eE]xit.*|.*[pP]rint.*). Prevents UI actions to be performed on UI elements whose *TITLE* (check appendix **??**) 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).
- **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.
- **Discount** = a decimal in the range 0..1. This parameter is used by the *qlearning* algorithm (check *TestGenerator* property).
- **DrawWidgetInfo** = true or false. Sets whether to display detailed overlay information, inside the *Spy mode* (see section 3.2), over the selected widget in the SUT' UI.
- **DrawWidgetTree** = true or false. Sets whether to display a graphical representation of the widget-tree, inside the *Spy mode* (see section 3.2), for the selected widget in the SUT' UI.
- **DrawWidgetUnderCursor** = true or false. Sets whether to display brief overlay information, inside the *Spy mode* (see section 3.2), over the selected widget in the SUT' UI.

[27]Sorted alphabeticaly

[28]Most can be edited through the tool User Interface, as described in section **??**. Nonetheless, you can edit them directly in the file.

[29]http://en.wikipedia.org/wiki/Regular_Expression

- **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.
- **GraphsActivated** = true or false. Sets whether to use the graphing feature of the tool (see appendix **??**).
- **GraphResuming** = true or false. If the *GraphsActivated* property is set to true, establishes whether to resume from a the last test sequence.
- **MaxReward** = a decimal greater or equal to 1. This parameter is used by the *qlearning* algorithm (check *TestGenerator* property).
- **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 (see section **??**).
- **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).
- **OfflineGraphConversion** = true or false. If the graphing feature is activated (check *GraphsActivated* property) and the *graphviz* was installed (check section **??**), sets whether to convert (false value) the graphs (*dot* to *svg*) at the end of a test. If the value is set to true, offline graph conversion can be performed from the graphs folder (check section 5.5).
- **OnlySaveFaultySequences** = true or false. Sets whether to save non-fail test sequences.
- **ProcessesToKillDuringTest** = regular expression (e.g. .*[oO]utlook.*|firefox.exe). 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).
- **PrologActivated** = true or false. Sets whether to calculate prolog based facts and rules representing information about the SUT (check the *prolog* algorithm at the *TestGenerator* property).

- **ProtocolClass** = settings_folder/Test_protocol_class_name. Links to the test protocol (see section 4) class under the folder denoted by the *MyClassPath* property.
- **ReplayRetryTime** = a positive number. Inside the replay mode (see section 3.3), 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[30]. Check the *StopGenerationOnFault*, *ForceToSequenceLength*, *MaxTime*, *SuspiciousTitles*, *TimeToFreeze* and *ProtocolClass* properties for specific behaviour.
- **Sequences** = a positive number. Number of times to repeat a test.
- **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[31], follow the next format: *web_ browser_ path SUT_ URL*.

  SUT_WINDOW_TITLE: *SUTConnectorValue* property must be the *title*[32] displayed in the SUT' 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 (e.g. .*[eE]rror.*|.*[eE]xception.*). Checks the UI for any suspicious title that could denote problems in the SUT. TESTAR checks whether there exists a widget' *TITLE* (check appendix **??**) 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 (see 5). For example, a critical message like "A

[30]Note: higher values will consume more hardware resources, specialy if graphing was activated.
[31]Check the *KNOWN_ ISSUES* file at tool installation folder (see section **??**) for compatible web browsers.
[32]Not applicable for empty titles

NullPointerException Exception has been thrown" can be represented by the regular expression ".*NullPointerException.*".

- **TestGenerator** = random, random+, qlearning, qlearning+, maxcoverage, prolog or evolutionary. Sets the UI action selection algorithm[33] during a test:

  random: picks a random UI action each time.

  random+: enhances random trying to jump to less explored UI.

  qlearning: a reinforcement learning approach[34] for UI actions selection.

  qlearning+: enhances qlearning trying to jump to less explored UI.

  maxcoverage: it tries to explore as much UI as possible.

  prolog: experimental. Uses prolog to decide on the next UI action to execute.

  evolutionary: experimental. Uses evolutionary computation to build a UI action selection algorithm.

- **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).

- **TypingTextsForExecutedAction** = a positive number. Sets how many typing actions with different texts must be performed, on the asme state' widget, to consider it an executed action.

- **UseRecordedActionDurationAndWaitTimeDuringReplay** = true or false. Inside the replay mode (see section 3.3) 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* (see section 3.2), for all the UI actions derived from the test set up (check section **??**).

The next set of properties must not be modified:

[33]It can determine the likelihood to reveal a SUT Fault. You can intervene in the selection at any time, switching to the *GenerateManual* working mode (check section **??**) through Shift + Left/Right-arrows keyboard shortcuts, from which you can perform the actions manually. Note that you will have to wait *ActionWaitTime* property seconds between actions, otherwise the actions will be missed in the recorded test sequence. To resume the automated tests switch again to the previous working mode.

[34]https://en.wikipedia.org/wiki/Q-learning

- **MyClassPath** = ./settings
- **OutputDir** = ./output -The directory where tests information can be found (see section 5).
- **TempDir** = ./output/temp -TESTAR will use this directory to store temporary files during the execution of test sequences.
- **PathToReplaySequence** = ./output/temp
- **FaultThreshold** = 0.01. Any verdict between this threshold and 0.9 will be considered a test fail.
- **ExecuteActions** = true or false. Unused property.
- **VisualizeSelectedAction** = true or false. Unused property.
- **ShowSettingsAfterTest** = true or false. Unused property.

E Keyboard shortcuts

Several keyboard shortcuts are available for the different working modes[35] (section **??**):

Keyboard shortcut	**Effect**	**Working modes**		
		Spy	Test	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	✓	✓(in debug mode)	
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 5: Keyboard shortcuts

[35]Check figure 24 for a flowchart of the different working modes and how to switch between them

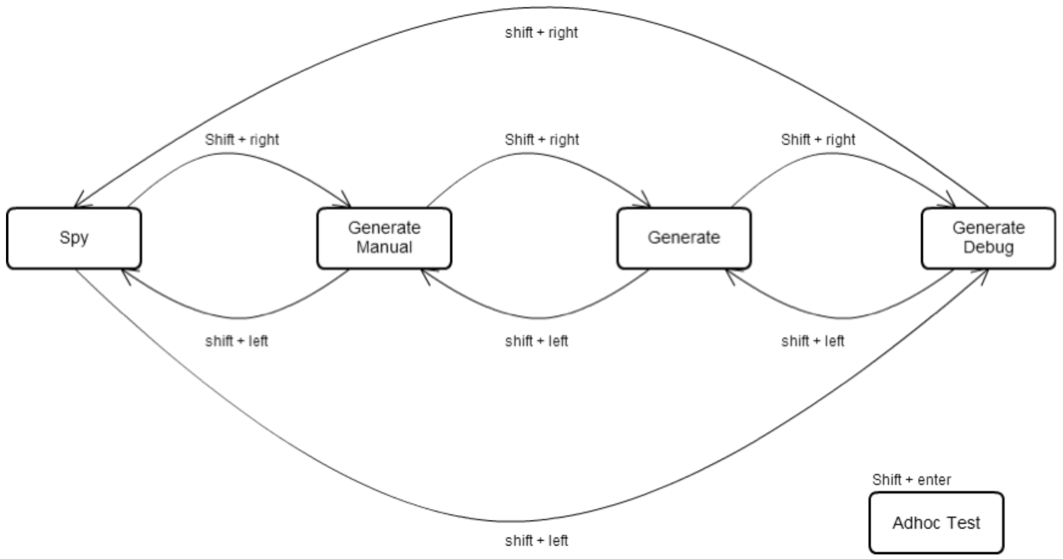


Figure 24: Working modes flowchart

F Directories

Key[36] tool directories are presented next:

./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
./suts	Sample SUTs binaries

Table 6: Directories

[36] *output* directory, and its contents, must exist before running tests. Refer to script *clean-output.bat*, which recreates the required directories. Be careful, you should previously copy/backup your tests reports (e.g. rename *output* to *my_ reports_ timestamp)*