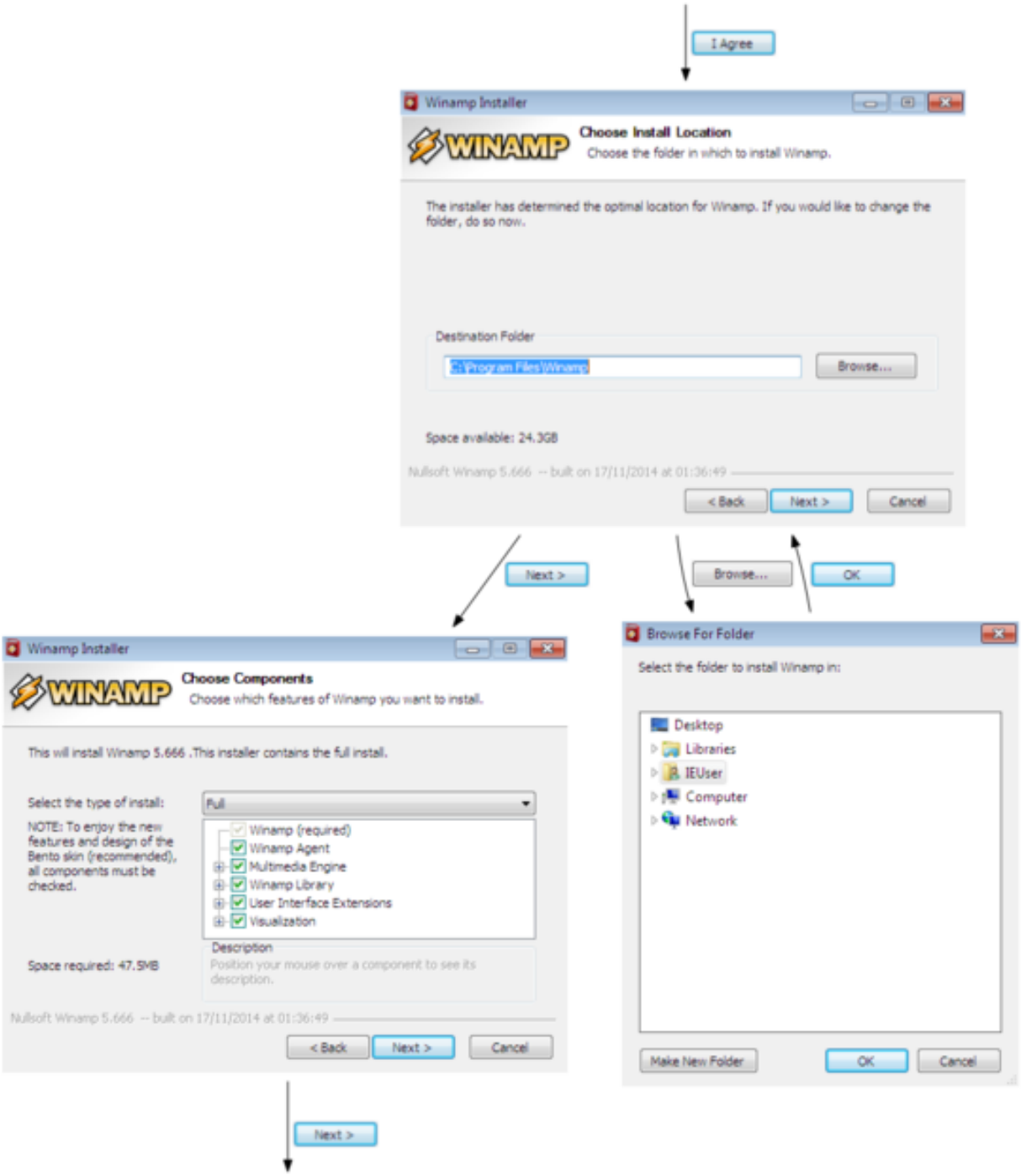
mrmurphy Documentation

Release 1.0.0

Matteo Cafasso

Jun 08, 2020

Contents



Framework for automating Graphical User Interface (GUI) based application analysis and testing.

CHAPTER 1

Rationale

Most of the available UI automation frameworks are built on the assumption the application to automate and the steps to perform are well known.

Mr. Murphy was originally built to explore complex GUI applications in order to find issues which other testing approaches could not. As it proved quite successful, it was soon tested against other use cases such as the automation of generic Windows installers as well as analysis of Potentially Unwanted Applications (PUA).

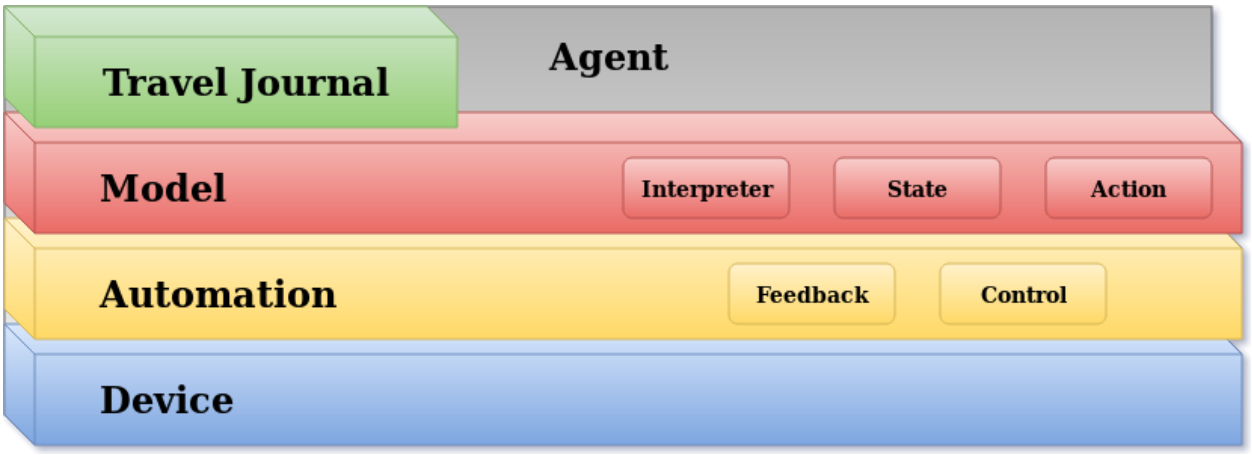
Though Mr. Murphy has a limited support for platforms and GUIs, its APIs were designed to enable easy extension of the framework.

CHAPTER 2

Design Principles

Mr. Murphy borrows several concepts and design patterns from basic Automation and State/Model representation principles.

The problem abstraction is divided in different layers which are here represented.



At the lowest layer, the Device can be either a physical computer, a mobile device or a virtual machine. The configuration, provisioning and operation of the Device are out of Mr. Murphy scope.

The Automation layer is the first abstraction implemented by the framework. Its responsibility is to enable the interfacing with the chosen Device. The Automation facilities allow to acquire raw data such as screenshots and control the Device interfaces such as mouse and keyboard. This layer can be extended to provide support to new platforms such as hardware devices or hypervisors.

The Model layer is responsible for interpreting the GUI state and providing a formal representation of it. A State consists of static information such as the window image and actionable items leading to the transition to a new State. This layer abstracts Operating Systems and windowing frameworks details.

The Travel Journal layer allows to annotate the State transitions and represent them as a directed graph. As the application under test might be unknown or presenting unexpected variations, the Journal becomes a useful support for the exploration process.

The last layer, the Agent, drives the execution of the GUI application. It does so by interpreting the application GUI State and comparing it with the information stored in the Journal. Agents implementations are Use Case specific. Few examples are available in the *agents* folder and can be used via the command line tool.

```
$ mrmurphy --help
```

CHAPTER 3

Resources

3.1 Set Up

3.1.1 Requirements

- Python > 3.5
- A virtualization capable host
- Libvirt and a supported hypervisor driver

  or

- Virtualbox

3.1.2 Installation

Please refer to your OS type and distribution manuals for setting up the chosen virtualization technology.

MrMurphy is available on PyPi.

```
$ pip install mrmurphy
```

3.1.3 Environment Setup

As Mr. Murphy tries to be the least invasive as possible, very little setup is required on the target environment.

The setup of the test environment depends on factors such as the OS platform and the type of tests which need to be performed.

The general recommendation is to reduce as much as possible the noise that other services and applications might generate. Minimizing the chance of undesired windows which could pop-up during the test execution will make the tests faster and more reliable.

Windows

Here follows a list of Microsoft Windows requirements and known limitations.

The *Windows Update* service is known to be problematic with Mr. Murphy as it might forcefully get control of the UI or reboot the device. It is derefore, recommended to disable it during the execution of the tests.

The current implementation cannot deal with higher priviledge desktops such as the *User Account Control* (UAC) prompt.

Scrapers

Mr. Murphy relies on a scraper software in order to retrieve a formal description of Windows GUI contents.

There are two implementations availables in *resources/scrapers*. The *uiauto* solution is the recommended one as it supports modern GUI frameworks.

The scrapers are meant to be run within the same environment of the tested GUI applications. They implement a simple HTTP protocol allowing Mr. Murphy to retrieve the application GUI content over the network.

The User must ensure the scraper is executing within the test environment and is reachable over the network.

The simplest of the solutions would be to place a script in the Windows *startup* folder to ensure the scraper is executed at boot time.

The following Visual Basic Script can be used to start the scraper hiding the prompt window.

```
Set oShell = CreateObject ("Wscript.Shell")
Dim strArgs
strArgs = "cmd /c C:\start_scraper.exe parameter other_parameter"
oShell.Run strArgs, 0, false
```

3.2 Agent Development

Agents are responsible for driving the execution of the GUI application. Therefore, their implementation vary according to the Use Case they are addressing.

The following chapter introduces a simple example of an Agent which interacts with every available object in a Windows application at most once.

3.2.1 Preparing the Interpreter

The Interpreter is responsible for constructing the GUI State and its available Actions. Interpreters offer a generic interface but their implementation depends on the Automation components which are provided. In this example, the *libvirt* and *vnc* based Automation interface will be used. Their details can be found in *murphy/automation*.

First, the *Feedback* and *Control* interfaces are built using their factory classes.

```
from murphy.automation.control import LibvirtControl
from murphy.automation.feedback import LibvirtFeedback

domain_id = "libvirt-vm-name"  # libvirt Domain ID, name or UUID
vnc_socket = "localhost:5900"  # VNC connection socket or address
```

(continues on next page)

(continued from previous page)

```
control = LibvirtControl(vnc_socket, domain_id)
feedback = LibvirtFeedback(vnc_socket, domain_id)
```

Secondly, the Windows scraper client is initialized. Mr. Murphy relies on a scraping agent running within the guest OS in order to de-construct the GUI content of a Windows application.

```
from murphy.model.scrapers.uiauto import WinUIAutomationScraper

scraper_address = "192.168.22.32"  # address of the scraper agent within the guest OS
scraper_address = 8000             # port of the scraper agent within the guest OS

scraper = WinUIAutomationScraper(scraper_address, scraper_port)
```

Lastly, the Windows Interpreter class is constructed using the above interfaces.

```
from murphy.model.interpreters.windows import WindowsInterpreter

interpreter = WindowsInterpreter(feedback, control, scraper)
```

3.2.2 The Travel Journal

The Travel Journal provides useful information regarding the execution of the GUI application.

```
from murphy.journal import Journal

journal_path = '.'  # local path where to store Journal information

journal = Journal(journal_path)
```

3.2.3 Exploring the application

Once all the resources are initialized, the control can be handed over to the main loop which implements the application exploration logic. The logic can be summarised as follows:

1. Interpret the current visible state of the GUI application
2. Compare the given state with the known ones in the Journal
3. Select a new action to perform
4. Perform the action
5. Return to point #1

```
import time

while True:
    new_state = interpreter.interpret_state()

    node = update_journal(new_state)

    action = choose_action()

    action.perform()
```

(continues on next page)

(continued from previous page)

```
    time.sleep(3)
```

Once a new state is returned by the Interpreter, the logic compares it with the Journal content. The Journal will return an old node or a new one according to whether the new state was ever encountered before.

The *update_journal* function sets the new node as the current one and renders the Journal as an HTML page.

```
def update_journal(state):
    if state in journal:
        node = journal.find_node(state)
    else:
        node = journal.new_node(state)

    journal.current_node = node
    journal.render()

    return node
```

The application State the Node refers to can be accessed from its namesake attribute. The State encapsulates the available Actions which can be performed. In order to acknowledge whether the Action was already performed, a boolean attribute is appended to its instance.

The action to perform is randomly chosen among the ones which have not been performed yet.

```
import random

def choose_action(node):
    actions = []

    for action in node.state.actions:
        if not hasattr(action, 'performed'):
            action.performed = False
            actions.append(action)
        elif not action.performed:
            actions.append(action)

    action = random.choose(actions)
    action.performed = True

    return action
```

3.2.4 Conclusions and references

The above implemented Agent is intended to be simple to give a complete yet easy to understand illustration of Mr. Murphy abstraction layers and their integration.

Several aspects such as the handling of errors and corner cases where purposedly omitted. This Agent implementation will wander aimlessly within a GUI application until it finds a State with no available Actions to perform and then it will crash.

More complete examples can be found in the folder *murphy/agents*.

CHAPTER 4

Interfaces & API documentation

4.1 Mr. Murphy

4.1.1 Mr. Murphy Automation

Facilities for interacting with a Device.

A Device can be either a physical computer, a mobile device or a virtual machine. The APIs are designed to be agnostic to the type of Device they interact with.

Interfaces

The interface API contracts can be found in the module *automation/interfaces.py*.

The interfaces are grouped in two categories: Feedback and Control.

Feedback

The Feedback class provides facilities for reading the state of a Device in order to interpret the execution of the GUI software.

The Feedback class abstracts interfaces such as the Screen and the Device Load Average.

Control

The Control class allows to alter the state of the Device to drive the execution of the GUI software.

The Control class abstracts interfaces such as generic input devices (Mouse and Keyboard) and execution checkpoints.

Factory interface

Available Implementations

Additionally, the automation module offers the following implementations of the above mentioned interfaces.

Feedback and Control

Subclasses

VNC Implementations

Device control facilities provided via vncdotool.

Note: VirtualBox does not support natively VNC. A VNC extension is provided but needs to be manually set up.

Module Content

Libvirt Implementations

Device control facilities provided via libvirt.

Module Content

4.1.2 Mr. Murphy Model

The *model* module abstracts the GUI application state providing mechanisms for analysing and changing it.

A *State* comprises of a *Window* and a list of *Actions* to be performed. States should implement comparison mechanisms to help tracing the execution of the application.

Interfaces

The interface API contracts can be found in the module *model/interfaces.py*.

Action types

Available Implementations

Additionally, the automation module offers the following implementations of the above mentioned interfaces.

Mr. Murphy Model Implementations

Windows

Scrapers

The *scraper* submodule contains the client interfaces and implementations for scraping the UI content of applications.

Mr. Murphy Scrapers

Scrapers client APIs. For the servers, check in *resources/scrapers/*.

Interfaces

The interface API contracts can be found in the module *model/scrapers/interfaces.py*.

Implementations

Windows UI Automation

Windows Native APIs

4.1.3 Mr. Murphy Travel Journal

The *Travel Journal* provides means for tracking the *State* changes the application undergoes during its execution.

Journal

Node

Edge

CHAPTER 5

Indices and tables

- genindex
- modindex
- search