SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2021;**31**:e1771
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1771

SPECIAL ISSUE PAPER

TESTAR – scriptless testing through graphical user interface

Tanja E. J. Vos[1,2,*,†], Pekka Aho[1], Fernando Pastor Ricos[2], Olivia Rodriguez-Valdes[1] and Ad Mulders[1]

[1]*Beta Faculteit, Open Universiteit, Heerlen, The Netherlands*
[2]*Centro de Investigación PROS, Universidad Politécnica de Valencia, Valencia, Spain*

SUMMARY

Covering all the possible paths of the graphical user interface (GUI) with test scripts would take too much effort and result in serious maintenance issues. We propose complementing scripted testing with scriptless test automation using the open-source TESTAR tool. This paper gives a comprehensive overview of TESTAR and its latest extensions together with the ongoing and future research. With this paper, we hope we can help and encourage other researchers to use TESTAR for their GUI testing-related research and pave the way for an international research agenda in GUI testing built upon stable and open-source infrastructure. © 2021 The Authors. Software Testing, Verification & Reliability published by John Wiley & Sons Ltd.

Revised 12 November 2020; Accepted 12 January 2021

KEY WORDS: test automation; graphical user interface; model inference; monkey testing; artificial intelligence

1. INTRODUCTION

The world around us is strongly connected through software, and our daily lives have become dependent on software. Consequently, software failures are having more and more impact. In 2017, the Software Fail Watch [1] reported that over 1.7 trillion USD losses caused by software failures and that more than 3.7 billion people were affected by these software failure. This study confirms the urgent need for correct and reliable software systems. Testing is critical, and one of the most used, activities for quality assurance whose goal is to address this need. This paper is about testing software systems through their graphical user interface (GUI).

The widespread use of iterative and incremental processes and continuous integration (CI) practices in software development has shortened the development cycles, drastically limiting the time for testing and quality assurance of each release. Instead of months or weeks, the longest period for testing a release is over a weekend or a night. The results of an automated smoke test set are expected almost instantly or in a few minutes. In practice, the use of test automation is a requirement for a successful CI process.

At the same time, the software systems are getting ever more complex, systems of systems with multitude of platforms and devices to support. The complexity of the systems being developed makes also software testing more difficult. A major part of test automation has been concentrating

*Correspondence to: Tanja E. J. Vos, Centro de Investigación PROS, Universidad Politécnica de Valencia, Camino de vera s/n, 46022 Valencia, Spain.
†E-mail: info@testar.org

on automating the execution of test cases. The combination of shorter time for testing and more software to test adds pressure to develop testing methods and tools that are more intelligent and efficient, reducing the manual effort from all phases of the testing.

The execution of unit tests is widely automated, and there are tools for unit test generation, such as EvoSuite [2,3], but system level testing is more challenging to automate, especially if the system includes GUI for the end users [4]. Testing software applications through their GUI is important because it can reveal subtle and annoying bugs. It is also expensive [5] and challenging, not only because of the combinatorial explosion in the number of event sequences but also because of the difficulty to cover the large number of data values [6]. GUIs represent the main connection point between a software's components and its end users and can be found in most modern applications. This makes them attractive for testers, because testing at the GUI level means testing from the user's perspective and is thus the ultimate way of verifying a program's correct behaviour [7]. GUIs are often large, complex and difficult to access programmatically, posing great challenges for testability of an application [8] and often resulting the testing process being carried out manually, which is an expensive and laborious task.

Manually recording or writing test scripts for all the possible paths of the GUI takes simply too much effort to be practical, and even if the test cases are built with keywords and proper architecture, so many test scripts would result in serious maintenance issues [9-13]. Obviously, there is a need to complement the script-based approach with another automated way of testing through GUIs.

In this paper, we will give a comprehensive introduction into TESTAR[1], an open-source[2] tool that implements a scriptless approach for completely automated test generation for web and Windows desktop applications. TESTAR is based on agents that implement various action selection mechanisms and test oracles. The underlying principles are very simple: generate test sequences of (state, action) pairs by starting up the system under test (SUT) in its initial state and continuously selecting an action to bring the SUT into another state. The action selection characterizes the fundamental challenge of intelligent systems: what to do next. The difficult part is optimizing the action selection [14] to find faults and recognizing a faulty state when it is found [15-17]. Faulty states are not restricted to errors in functionality; also, violations of other quality characteristics, like accessibility or security, can be detected by inspecting the state. This totally shifts the paradigm of GUI testing: from developing scripts to developing intelligent artificial intelligence-enabled agents.

The development of TESTAR started during the FITTEST project [18] that run from 2010 to 2013. After that, the development continued while piloting the tool in various companies in ERASMUS+ projects like SHIP [19] and various nationally funded projects from the Spanish and Valencian governments. In 2017, the development was continued in the context of the TESTOMAT ITEA3 project. The first overview paper has been published [20] in 2015. Over the past 4 years, TESTAR has been developed and extended significantly, and many parts of TESTAR have been changed. We have received many requests to write up a complete overview of how TESTAR works together with the state of the art, related work and future research directions. This paper is our reply to all of these requests and compiles and extends 10 years of research on the tool.

On the road towards a tool that can learn by itself how to test a system, TESTAR has been extended with a memory in form of state model inference, and intelligence, for example, by using machine learning.

The novel extensions of TESTAR described in this paper include (i) active learning of state models (model inference) described in Section 5; (ii) support for using Selenium WebDriver instead of Windows Accessibility API for testing web applications; and (iii) integration into CI processes, and generating HTML reports. In addition, we report on new (industrial) empirical studies evaluating the test efficiency and effectiveness.

Finally, combining the results of all studies that were done over the years, we find that TESTAR constitutes a valuable complementary testing tool for manual testing and scripted GUI test automation. With this paper, we hope we can encourage people to use TESTAR, not only for testing their software at the GUI level in real industrial environments but also for their GUI testing-related research, and pave the way for an international research agenda in GUI testing that can be build upon

[1]https://testar.org/
[2]https://github.com/TESTARtool/TESTAR_dev

DOI: 10.1002/stvr

stable and open-source infrastructure. This enables researchers to concentrate on intelligent testing techniques while reusing existing automation facilities.

The rest of this paper is structured as follows. Section 2 describes the state of the art of script-based GUI testing, while Section 3 describes scriptless test monkeys. In Section 4, we describe the basic functionality of the TESTAR tool. Section 5 describes how we add memory to the TESTAR monkey by inferring a state model during the automated GUI exploration. Section 6 describes different ways to add intelligence to the monkey while selecting actions. We will describe how actions can be filtered to concentrate on the important ones, or how to make action derivation smarter. Moreover, the section describes how action selection strategies can be improved using reinforcement learning, ant colony optimization (ACO) and meta-heuristics. In Section 7, we describe for each industrial case study the company, the SUT, the context, the objectives and the results of the studies. Section 8 summarizes the ongoing and future research directions that have been put forward throughout this paper, and Section 9 gives the final conclusions.

2. SCRIPT-BASED GRAPHICAL USER INTERFACE TEST AUTOMATION

Automated GUI testing has been classified in various ways. In [21], we can find a classification based on how the test automation tool interacts with the SUT. This results in a classification of three generations: the first is based on the mouse coordinates, the second is based on technical APIs and the third is based on image recognition. A more recent classification was described on [22] that extends the three generations with another axis addressing the level of automation. In this section, we will follow the classification from [22] to describe the state of the art of script-based GUI testing.

2.1. Manually recorded or written scripts

Testing through GUIs is commonly automated by scripts that are captured or manually created with a script editor, automating the execution of test cases. A major challenge with script-based GUI test automation is the manual effort required for maintaining the scripts when the GUI changes [11]. Test script maintenance has been a challenge for a long time [12], but it is still a relevant problem in the industry. Usually, scripts are created according to a use case with the corresponding input sequence and later automatically replayed on the GUI to serve as regression tests.

With capture and replay (CR) tools, for example, commercial tools like Rapise [23], Squish [24] and Ranorex [25], the scripts are recorded during manual use of the SUT through its GUI. Then the recorded scripts can be automatically executed on another version of the same GUI to detect changes in the behaviour. The test oracle is the presumably correct behaviour of the recorded version. The advantage of CR is that it does not require special skills to use. Recording of the test cases is very similar to manual GUI testing. CR techniques differ mainly in the kinds of GUI technology they can handle [5], but more advanced CR tools provide a graphical editor to make it easier to change and maintain the recorded scripts.

With other script-based tools, the test steps of the test scripts are manually defined with some scripting language that can be textual, as with AutoIt [26], or the tool provides a graphical editor for scripting, as with SeleniumHQ [27], and Visual GUI Testing tools SikuliX [28] and EyeAutomate [29].

The main challenge in industrial adoption of script-based GUI testing is the maintenance effort required when the GUI changes [11]. With CR tools, any test sequence going through the changed parts of the GUI might have to be manually recorded again. With scripting languages, any script going through the changed parts might have to be manually updated. The more advanced tools are better resistant to small GUI changes, but a bigger change in the GUI will cause the test scripts to fail, resulting false positives until the scripts have been repaired. The maintenance effort greatly diminishes the return of investment of script-based GUI test automation. Another challenge is the effort required to create the test scripts in the first place. Manually written test scripts are shown to be easier to maintain but require more effort to create than recorded scripts [13]. Test scripts tend to be executed and maintained a lot during a project [12]. Therefore, the one-time investment in the beginning should not be as big a problem as the continuous maintenance.

DOI: 10.1002/stvr

If the test scripts are manually defined or the recorded scripts are manually elaborated, it is possible to define test oracles that check the correctness of any value in any specific state of the GUI. However, each check has to be defined separately, and obviously, this increases the manually defined test artefacts that have to be maintained when the GUI changes. Most script-based GUI testing approaches check only if the most important data values are correct in a specific state of the GUI. Manually checking all the values of all the properties of all the widgets of each GUI state is not feasible even without the maintenance effort, but setting too few values may lead to ambiguous test verdicts [30].

There are academic research approaches on automated GUI script repair to tackle this maintenance problem, for example, WATER [31], VISTA [32] and SITAR [33], but none of them have been widely adopted in the industry. Some commercial tools, for example, Squish [24], claim to update the test scripts automatically when the GUI changes, but no details or data are given whether this works in practice and how. There are also approaches aiming to make the scripts more robust against the SUT changes, for example, ROBULA+, an algorithm for generating more robust XPath-based locators [34].

2.2. Test case generation based on manually created models

A more advanced approach, as compared with the CR tools, is model-based GUI testing (MBGT). It requires that a model of the GUI and its expected behaviour is created, which has a higher level of abstraction than the GUI itself. The modelling language should be understandable by a tool that will take this model and use it to automatically generate tests. An advantage of this type of testing is that one can quite formally and precisely specify the exact specifications that a GUI should conform to. Consequently, the expected behaviour captured in the model enables generating system-specific automated test oracles. The intended benefit of MBGT is that when an application's GUI changes, manual update of all the test scripts is no longer needed. Instead, the model is updated and the scripts/tests are automatically generated again. The disadvantages are that one has to have a deep knowledge of the application domain that the GUI covers, one has to have fairly expert knowledge of formal modelling methods and languages and, finally, that manually creating such a model is time consuming.

There are various MBGT approaches, for example, using Spec Explorer in a tool chain [35], NModel tool [36], pattern-based GUI testing [37], Finite State Machine (FSMs) to test web applications [38] and TEMA tools for testing Android applications [39], trying to reduce both the initial effort in creating the test scripts and the maintenance effort required after each change in the GUI. The difference to the script-based testing is that in model-based approaches the scripts are automatically generated. There are also various kind of models used for MBGT, for example, state-based models like finite state machines [40] and event-based models like event flow graphs [41].

The challenge in MBGT is the specialized formal expertise and effort required for creating the models that allow automated test case generation and execution on real-life GUI applications [42]. Designing a test model on a suitable level of abstraction, and following the exact modelling syntax supported by the test case generation tool, is not a trivial task. Another challenge is that usually a test adapter has to be developed for each SUT, and the model has to be mapped into the implemented functions of the adapter, so that the generated test cases can be executed on the SUT.

2.3. Test case generation based on inferred models

Graphical user interface ripping [41] and other GUI model extraction or inference approaches, for example, GUI Driver [43], Crawljax [44] and GuiTam [40], try to help in creating the GUI models for testing. There have been some static approaches based on source code analysis, but it is difficult to capture the dynamic behaviour of the GUI without executing it. Most dynamic approaches, for example, GUI Driver [43] and Extended Ripper [45], analyse the behaviour of the GUI during run-time while emulating the end user by automatically interacting with the GUI widgets to traverse through the GUI. Some approaches, for example, [46], combine dynamic and static analyses for model extraction. During model extraction, it is possible to use generic checks to detect failures, such as unhandled exceptions and crashes, which is actually scriptless GUI testing.

DOI: 10.1002/stvr

Many approaches, for example, [40,41,43,46], have used the extracted models for generating test sequences. By executing the generated test sequences on another version of the same GUI, it is possible to detect changes and regression bugs between the SUT versions, in addition to generic failure checks. When generating test cases from a model extracted from previous GUI version and executing them on the new version, the challenge is that the test cases will not test or notice any new parts of the GUI. The new parts were not in the extracted model, so they cannot be in the generated test cases either. Test cases fail when something included in the test cases is removed or changed. Usually, it is easy to extract a new model from the new version, but then we lose the reference behaviour that is used to discover the changes.

3. SCRIPTLESS GRAPHICAL USER INTERFACE TEST AUTOMATION

Monkey testing (also called random testing or stochastic testing) refers to a scriptless approach of randomly exercising a SUT by means of an automated test tool [47]. Unlike in script-based testing, test cases are generated each time during testing. Often, the generated test cases are not even saved, because in scriptless testing we are usually interested only in sequences that find failures. This is the reason why scriptless testing does not require test case maintenance. Depending on the action selection strategy, most test monkeys explore the SUT in a different way each time the test is run, consequently making it possible to find new defects in the same SUT version by just executing more tests. Usually, test monkeys perform random testing by issuing clicks and keystrokes in order to crash the GUI or render it unresponsive. The literature distinguishes two main types of test monkeys [47,48]: (i) dumb monkeys, which do not possess any knowledge about how to use the GUI and usually click and type entirely at random, and (ii) smart monkeys, which have a basic understanding of the GUI and often employ a model that guides them and helps to make decisions as to which actions to execute.

3.1. Dumb test monkeys

The pure dumb or ignorant monkeys have no idea what state the SUT is in. They do not know what inputs are legal or illegal, nor what a failure is. Still, there are various ways to increase the intelligence of a monkey, and even though it can still be ignorant about your SUT, it can understand its environment and can find very obvious bugs like crashes and hangs. Such tools have been used since the early 80s at big companies like Apple [49] and Microsoft [48]. According to Nyman [48], in some Microsoft applications groups, 10–20% of the faults in their projects are revealed by test monkeys. In addition, Microsoft actively recommends randomized techniques for safety critical applications [50]. Also, Google has developed *The Monkey* [51] for testing Android applications. Monkeys often find severe faults that may be caused by obscure sequences that are hard to figure out, even for experienced testers. The fact that these tools operate completely automatic makes their application cheap and thus attractive.

Not all researchers, however, are convinced of the effectiveness and usefulness of random testing, and the subject has been controversial throughout the history. Girard and Rault (1973) call it a *valuable test case generation scheme* [52]. This is confirmed by Thayer *et al*. (1978) in their book on software reliability [53]; they say it is the *necessary final step in the testing activities*. However, Glenford Myers (1979) in his seminal work on the art of software testing [54] denominates random testing as *probably the poorest testing method*. Beizer [55] refers to this kind of testing as 'keyboard-scrabbling' and advertises other techniques.

In 1984, however, Duran and Ntafos [56] carried out a series of experiments in which they showed that random testing could be more effective than the commonly used partition testing. Hamlet and Taylor [57] repeated more experiments and came to the same results. Weyuker together with Jeng compared the two testing approaches from an analytical point of view [58]. However, their results pointed in the same direction again: a clear superiority of partition testing could not be stated; instead, it turned out that, in effectiveness, partition testing can be better, worse or the same as random testing, depending on the 'adequacy' of the chosen partition with respect to the location of the failure-causing inputs.

DOI: 10.1002/stvr

These were counter-intuitive results that opened the doors to a large body of literature on the properties and benefits of random testing. Many authors investigated the conditions under which partition testing can be more effective than random testing or the other way around (see, e.g. [59-62]).

Some more recent studies show that we have by no means investigated enough on random testing. Arcuri *et al*. [63] show that random testing is an instance of the *coupons collector problem*, a very well-known probabilistic problem. This way, many theoretical results from the probability field can be reused, and again, it is shown that random testing is not a bad testing strategy in many occasions.

Böhme and Paul [64] present a study analysing the efficiency of random testing. Basically, they conclude that *even the most effective testing technique is inefficient compared with random testing if generating a test case takes relatively too long*.

Amalfitano *et al*. [65] defined a framework for comparing testing techniques. As part of their research, effectiveness and cost of random and smarter techniques were compared. In order to measure effectiveness and cost, they used code coverage and the number of overall iterations until the technique reaches its termination point, respectively. While in most cases the smarter techniques were less expensive in terms of iterations, in all cases, the random exploration strategies showed greater effectiveness, that is, code coverage.

However, there are also fundamental weaknesses. First of all, random testing requires a lot of execution time to get coverage. This can be partly solved by parallel execution, so with enough resources, it is possible to get the results faster. Another challenge is reaching all parts of the GUI. Often there are login screens or other parts that require specific input to reach. One option would be combining other tools with test scripts to reach a specific part of the GUI and then start the random testing.

An important challenge with random testing is the test oracle. Most random testing tools use only generic checks to find exceptions and crashes or getting the GUI into unresponsive state, not supporting manually defined application-specific test oracles or regression testing between SUT versions. These general test oracles are useful for robustness testing but limit the potential of random testing. If a monkey finds an error, then the length of the generated sequences and their reproducibility might become a problem [48]. Monkeys may run several days before they crash the application, which results in extremely long sequences of randomly distributed clicks. It is very difficult to reliably replay these sequences for debugging purposes.

3.2. Smart test monkeys

There are various ways to make a test monkey smarter. The first step is making the monkey aware of the environment. Bertolini *et al*. [66] found that the time needed to trigger a crash can be significantly reduced if the monkey is aware of the visible widgets and specifically targets them. This can be done, for example, using a technical API or a library to get a programmatic structure of the layout and the widgets of the GUI. Another way could be using image recognition to detect widgets from the screenshots.

The second step is using that information to deduct the state of the GUI. Then a monkey has a basic understanding of the GUI's widgets, knows how to use them and is capable of issuing complex actions in order to effectively drive the SUT. The monkey can then, for example, create sequences as follows: (i) determine the state of the GUI, that is, the visible widgets, their size, location and other properties (such as whether they are enabled or blocked by other windows), then (ii) derive a *set of feasible actions* from which it then (iii) selects one that it finally executes it. By repeating these steps, the monkey will be able to generate arbitrary input sequences to drive and hopefully crash the GUI.

Usually, to make monkeys smart enough to reach all parts of the GUI, they need to have some knowledge about the SUT. One way to give the monkeys more knowledge is through a state model of the SUT [48]. The smarter monkeys can then use the state model during traversal, i.e. they can choose from the legal actions in the current state for moving to another state and then verify that the next expected state has been reached. Illegal inputs can be added to the monkey's repertoire if the

DOI: 10.1002/stvr

model includes error-handling states. However, with manually crafted models, this goes a long way towards MBGT.

Another way of giving SUT-specific knowledge to the smart monkey is defining them with kind of agents that trigger into action when a specific state of the GUI has been detected. This approach is used, for example, in TESTAR tool [20] and Murphy tools [67], and some examples are given in Section 6.3.

Defining SUT-specific knowledge creates testwares that have to be maintained if the SUT changes. In some cases, the tool could deduct or learn how a specific application should be tested, using artificial intelligence or machine learning. Some examples are given in Section 6.4.

4. TESTAR

TESTAR[3] is an open-source tool that carries out automated testing without the need for scripts, falling into the category of smart monkey testing tools. It implements a scriptless approach, meaning that the test cases do not have to be defined prior to test execution. Instead, each test step is generated during the test execution, based on the actions that are available in that specific time and state of the GUI.

The underlying principle of TESTAR is very simple: generate test sequences of (state, action) pairs by starting up the SUT in its initial state and continuously select an action to bring the SUT in another state. The action selection characterizes the most basic problem of intelligent systems: *what to do next*. The difficult and challenging parts are optimizing the *action selection* to find faults and recognizing a faulty state when it is found with an *oracle*.

The high-level logical flow of TESTAR is illustrated in Figure 1. Everything inside the greybox is automated. The three activities on the right side of the box represent activities that testers can improve programmatically if desired (i.e. action definitions, filtering, selection and oracles). After starting up the SUT, the tool goes into the loop of continuously selecting and executing an *action* to bring the SUT from one *state* to another state, until some stopping criteria have been met, after which the SUT is closed. In the following sections, we will describe each of the basic steps of the approach:

- obtaining the GUI state (Section 4.1);
- deriving the set of actions that a potential user can execute in that specific state (Section 4.2);

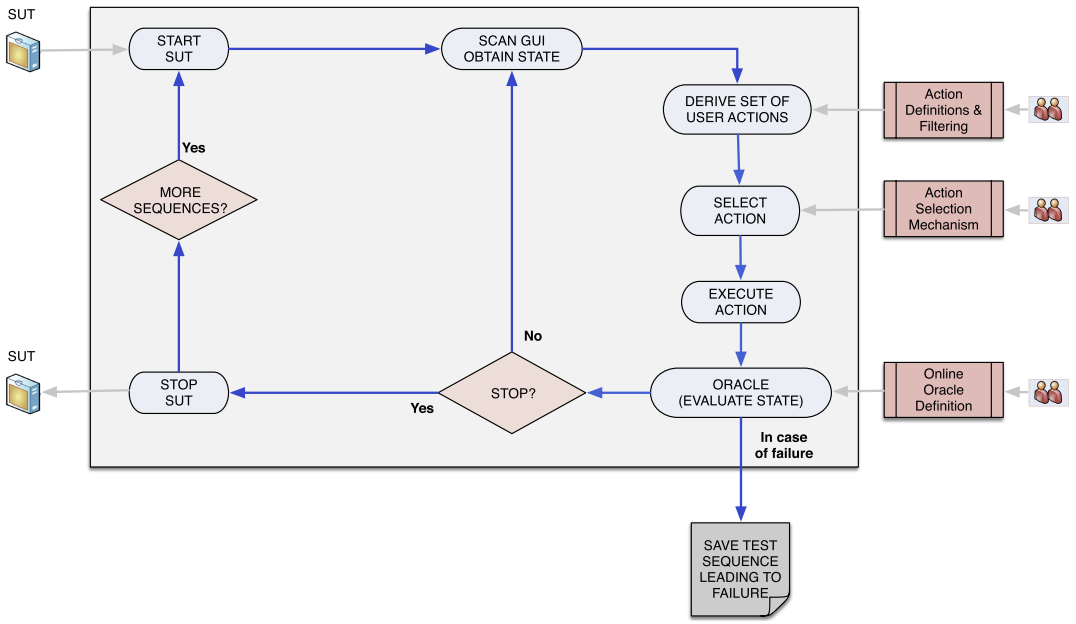

Figure 1. TESTAR testing cycle. GUI, graphical user interface; SUT, system under test.

DOI: 10.1002/stvr

- selecting and executing one of these actions (Section 4.3); and
- evaluating the new state to find failures (oracles) (Section 4.5).

Subsequently, in Section 4.6, we will describe the runtime execution of TESTAR and the execution of the test sequence loop (as shown in Figure 1). In Section 4.7, we will describe the results and outputs of a test run.

4.1. Obtaining the graphical user interface state

A GUI can consist of a wide range of *widgets*. Examples of these are in Table 1.

These widgets are structured in a hierarchy that is called the *widget tree*. Figures 2 and 3 display examples of such widget trees. Each node corresponds to a visible widget, contains widget properties like its type, position, size and title and indicates whether it is enabled and so forth. These trees and properties can be obtained automatically in various ways, for example, by using *accessibility APIs* – which allow computer usage for people with disabilities – at the operating system level (i.e. UIA Automation for Windows, ATK/SPI for Linux and NSAccessibility for MacOS). These accessibility APIs allow us to gather information about the visible widgets of an application and give TESTAR the means to query their property values. We can also use, for example, programmatic APIs, or *automation frameworks*, like the Selenium WebDriver [27] at the browser level, or Appium [68] for iOS, Android and Windows [68]. Moreover, we can use *language-specific APIs* like the Java Access Bridge for existing Java objects at the Java Virtual Machine level, or even *image recognition* [67]. Also, *SUT-specific APIs* are an option, a first experience with that has been done by testing a smart home through a RESTful API with TESTAR [69]. Image recognition can be used as a platform-independent way to obtain the state of the GUI from the screenshots, but at least the current image recognition algorithms are less accurate and give less information than the technical APIs.

In the current[4] implementation of TESTAR, there are plugins for detecting the state using the UIA Automation for Windows and the Selenium WebDriver. For example, the UIA API gives access to around 170 attributes or properties [70], which allows us to retrieve detailed information such as

- the **role** of a widget, is it a button, checkbox, drop-down and so forth;
- the **path** that the widget has in the stack of widgets on the screen, that is, the widget tree;
- the **size** that describes 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 **focused** (has keyboard focus) so that the monkey knows when it can type into text fields; and
- attributes such as **title, help** and other descriptive attributes are very important to distinguish widgets from each other and give them an identity.

All these properties and their values are stored in the *widget tree*. In this way, these trees capture the current *state s* of the GUI like the examples from Figures 2 and 3. Imagine we have a widget tree

Table 1. Examples of widgets of which a graphical user interface can be composed.

Windows	Menus	Controls
• Main window • Child windows • Pop-up windows • Dialog windows	• Menu bars • Drop-down menu • Context-aware menu	• Buttons • Text boxes • Links • Radio buttons • Checkboxes • Drop-down select boxes • Sliders • Tabs • Scroll bars

DOI: 10.1002/stvr

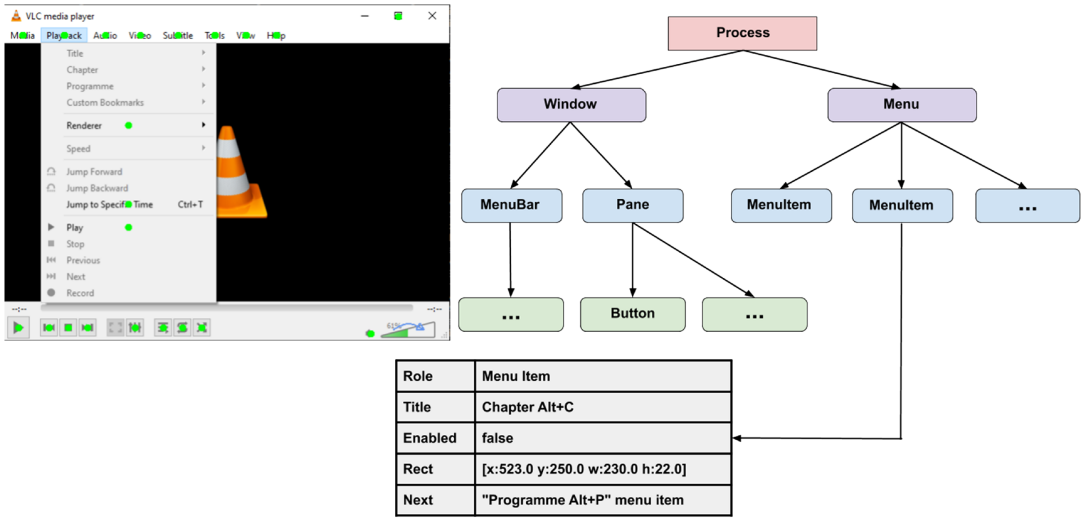

Figure 2. The state of a graphical user interface as a widget tree.

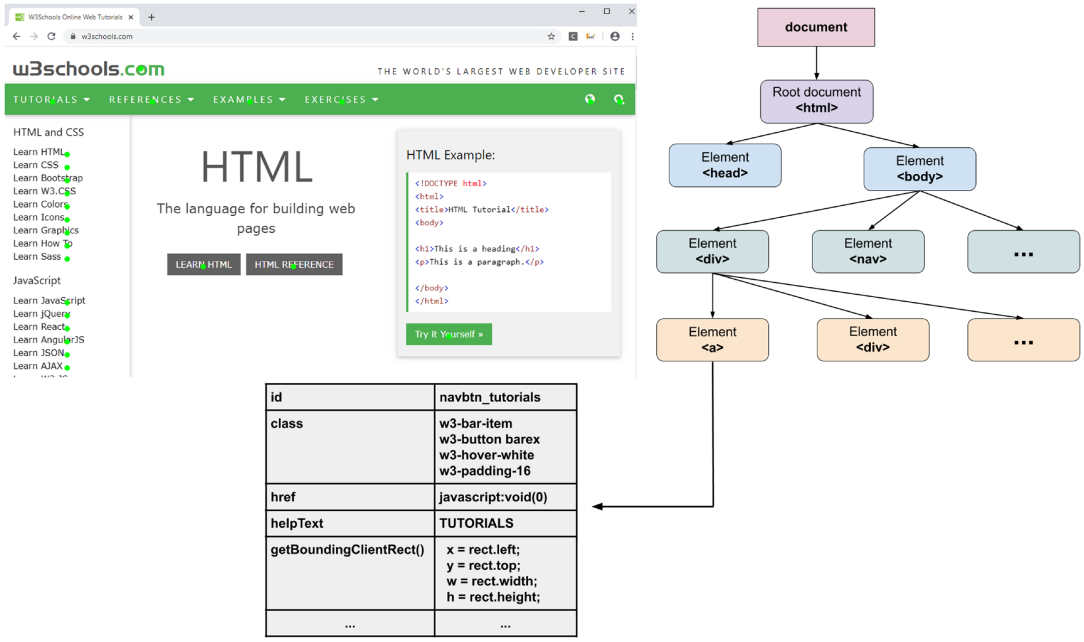

Figure 3. The state of a graphical user interface can be described as a widget tree.

that represents a specific state s. The nodes of the *widget tree* are the widgets that are visible on the GUI in that particular state s. We will denote this set of nodes with $W(s) = \{w_1, w_2, \ldots, w_k\}$ (e.g. button, slider, text field and menu). The edges of the tree reflect the parent–child relationships: each child widget is displayed within the screen area occupied by its parent widget. We will denote the set of edges with $E(s)$. Consequently, there exists a directed edge $(w_i, w_j) \in E(s)$ when $w_i \in W(s)$ is the parent widget of $w_j \in W(s)$ in state s. The values of the all properties that the widgets have also define the state. For a widget $w \in W(s)$, we will use $P(w, s)$ to denote the set of all properties $\{w.p_1, w.p_2, \ldots, w.p_m\}$ (e.g. **role**, **title**, **position**and **enabled**) for that widget w in state s.

All the properties $P(w, s)$ obtained by TESTAR in state s for the widgets in $W(s)$ through an API or an automation framework are associated to the TESTAR representation of `States`, `Widgets` and `Actions`. This is done through `Tags` and is depicted in Figure 4.

DOI: 10.1002/stvr

Taggable classes implement the `Taggable` interface, which means that `Tags` can be added to their instances. In TESTAR, the classes `State`, `Widget` and `Action` are taggable and the `Tags` are pairs of (property name, value). Properties that are common to all widgets are defined in a final class `Tags`. The properties specific to an implemented API technology or automation framework (like Windows UIAutomation, Selenium WebDriver and ATK/SPI) are defined in specific API-taggable final classes (`UIATags`, `WebTags`, `AtSpiTags` etc.). We can use the `get` method to read the properties of taggable objects (i.e. instance of the classes `State`, `Widget` and `Action`) as follows:

```
taggableObjectName.get(Tags.PropertyName)
```

In Example 1, there is an if statement in line 1 whose guard checks whether some `action`'s **role** tag equals `LeftClick`. Similarly, in line 3, the **href** tag is checked for some `example-text` that we want to act upon in the if statement.

Example 1: Obtaining property values

```
1 if (action.get(Tags.Role).equals("LeftClick"))
2     {.....}
3 if (widget.get(WebTags.Href).contains("example-text"))
4     {.....}
```

Obtaining the state, that is, extracting the properties of the widgets and building the widget tree, is done automatically after each executed action. For Windows desktop applications, TESTAR monitors the CPU usage of the SUT process to figure out when the SUT has finished executing a GUI action. However, sometimes the widget tree is extracted before the GUI has finished updating, resulting with a partial widget tree. If the partial tree contains interactive widgets, actions are derived for them. If not, a default action (such as executing a NOP action or pressing ESC key) will be executed and testing continues by deriving the state again. TESTAR can be configured to change the waiting time between the executed action and the next widget tree construction. For Web applications and the Selenium WebDriver framework, we offer the possibility to use a JavaScript command (`document.readyState`) to wait until the web page has been loaded. However, this has the disadvantage of having to wait for web pages to load their ads. Moreover, in collaboration with

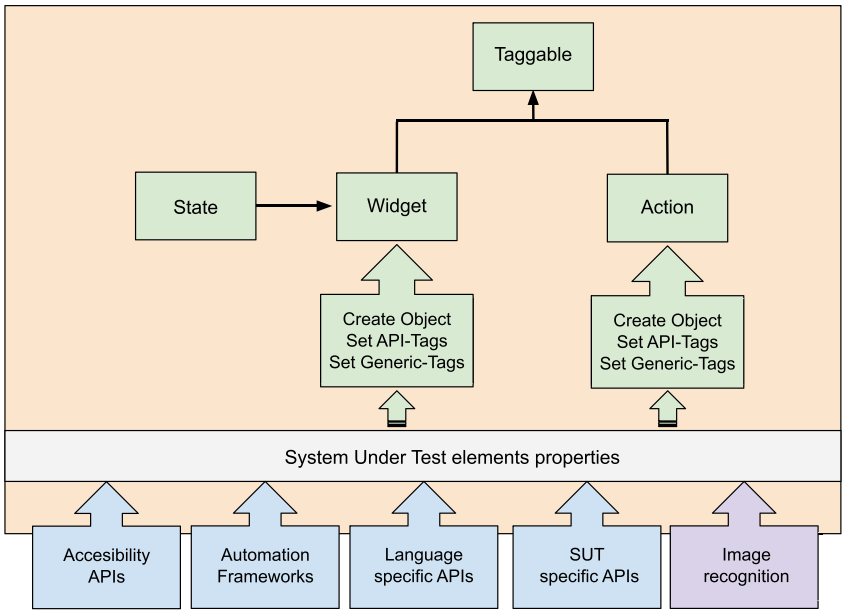

Figure 4. Taggable classes `State`, `Widget` and `Action`. SUT, system under test.

DOI: 10.1002/stvr

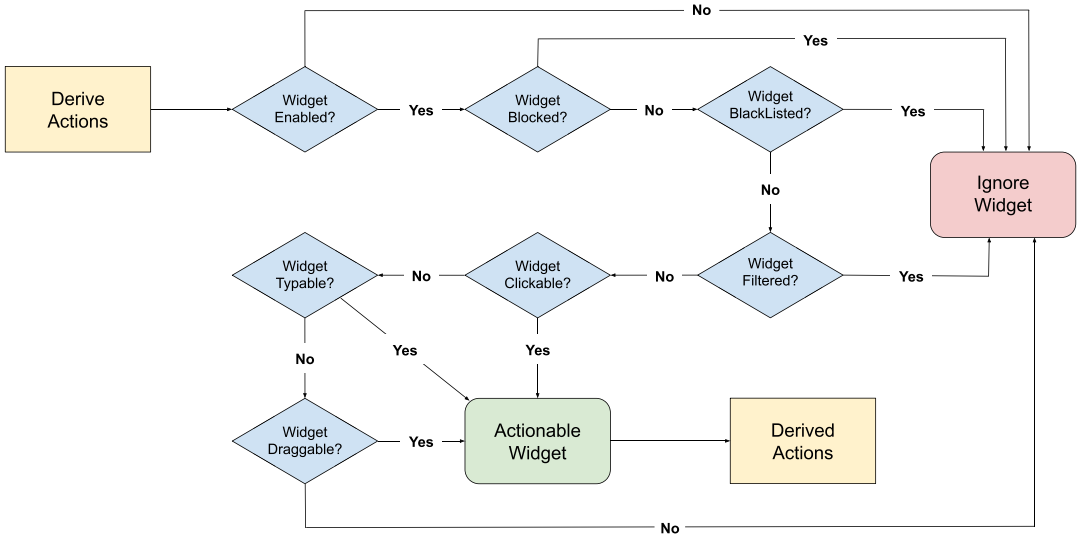

Figure 5. Deriving actions from actionable widgets.

some partners, we have found that this functionality is not enough in some cases, because the web document says it is ready, but the internal server is still processing data.

4.2. Deriving a set of actions

Having obtained the GUI's current state s, we can go on to derive a set of available actions that a user can choose from in that specific state that are suitable for most applications. For this, we first derive a set of *actionable widgets* (Figure 5). Actionable widgets are widgets on which we can act because they

- are enabled;
- are unblocked;
- are not blacklisted or filtered by a tester (Section 6.2); and
- expect user interaction, that is,
  - widgets that are clickable (left or right mouse button);
  - widgets that are typable; and
  - widgets that are draggable or slidable.

For example, suppose in state s we have a clickable button widget $\texttt{b} \in W(s)$ that is enabled and unblocked.[5] If a tester did not blacklist or filter this widget, then this means there exists a possible action that can click on that button (**click**(b)). Likewise for an actionable typeable text field widget $\texttt{t} \in W(s)$, it means there exists a possible action that can click to focus and type into that text field (**type_into**(t)).

To derive the actions that can be executed in a certain state s, TESTAR loops through the widget tree and collects those actionable widgets. To create executable actions from these actionable widgets, TESTAR converts them into implementations of the `Action` interface (Figure 4).

An execution scheme for button $\texttt{b} \in W(s)$ from above is as follows:

1 determine the `position` on the screen that falls inside the widget;
2 move the mouse cursor to that point;
3 press the mouse down; and
4 release the mouse.

The movement of the cursor and pressing and releasing the mouse button each have their own implementation of `Action`, called `MouseMove`, `MouseDown` and `MouseUp`, respectively.

[5]More specifically: `b.(Tags.Enabled) == true` and `b.(Tags.Blocked) == false`.

DOI: 10.1002/stvr

A fourth implementation of `Action` is introduced in the form of the `CompoundAction` class. This class aggregates sequences of actions into an `Action`. Example 2 shows how to create an action to click a button.

Example 2: Create action for button `b` ϵ W (s)

```
1  public Action leftClickAt(Position position){
2      return new CompoundAction.Builder()
3                  .add(new MouseMove(position), 1)
4                  .add(MouseDown, 0)
5                  .add(MouseUp, 0)
6                  .build();
7  }
```

In the case we obtain the current state using the Selenium WebDriver implementation, in addition to interact with the browser through the actions described earlier, we can also derive a set of actions that represent JavaScript commands that we can execute through the Selenium WebDriver interface (i.e. calling `WebDriver.executeScript(JScommand)`).

These JavaScript commands allow us to interact with the web elements that exist in the current web document. The Document Object Model (DOM) API is used to find the web elements together with their attributes or interact with the web document through the browser. TESTAR predefines a couple of JavaScript commands to define actions that are considered useful during testing. These are defined internally as calls to `WebDriver.executeScript`:

- `WdCloseTabAction` to close a tab;
- `WdHistoryBackAction` to simulate a click on the history back button in a browser;
- `WdSubmitAction` to simulate a click on a submit button in a detected web form; and
- `WdAttributeAction` to find a web element by its unique identifier and write a value in the desired attribute using a pair (key, value).

It is possible to change or add new actions on the tester's need (Section 6.3). As an example, let us consider `WdAttributeAction`. Inside a web document, a web element can be searched and retrieved using one of its web attribute. Then, with the focus on the desired web element, some DOM API web methods allow us to read or write a value in one of the multiple attributes of this web element. This is defined in `WdAttributeAction` as follows:

Example 3: Create a customWebDriver action using a JavaScript command and the `executeScript` interface

```
1  public WdAttributeAction(String elementId, String key, String value) {
2      WebDriver.executeScript(
3          String.format(
4              "document.getElementById('%s').setAttribute('%s','%s');",
5              elementId, key, value));
6  }
```

Other types of actions that are being derived are those related to force the SUT to be in the foreground or clean some undesired processes. To achieve the first, we use native calls intended to invoke the main window to the foreground. If this for some reason is not possible, we use keyboard commands as Alt + Tab. To kill undesired processes, we constantly check, after each action, the existing processes in our SUT's environment.

Moreover, with web applications, in addition to maintain the desktop browser in the foreground, we need to ensure that we do not lose the focus of the URL domain of the SUT we are testing and start exploring non-desired web pages.

We will denote the set of actions that are derived in state s by $A(s)$.

DOI: 10.1002/stvr

4.3. Select and execute one of these actions

So we are in state s where we have derived a set of actions $A(s)$ that can be executed. Now we can select one, say a, and execute it. In TESTAR's default mode, that means random selection. The action a gets executed, and we go to a new state s'. This way test sequences are generated like $s \rightarrow_a s' \rightarrow_{a'} s'' \rightarrow \ldots$ until some stopping condition holds. stopping conditions can be for example, when a failure was found, or when a configured amount of actions have been selected and the test sequence hence has reached its predefined length. For this, when starting up TESTAR, we can define how many sequences we want to generate (`number_of_sequences`) and how many actions we want to select for creating each sequence (`number_of_actions`).

4.4. Representation of states and actions

In order to be able to recognize and compare states and actions, we need to assign a unique and stable identifier to each of them. For this, we can use the attribute, or property, values that come with each widget within the widget tree in a specific state s. If we use all the properties, we obtain what we would call a *concrete* identifier. However, we do not need all to use all of them. We can select a subset and use *abstract* identifiers instead. When selecting properties for the identifier we should take care that they are relatively stable properties. For example, the **title** of a window is quite often not a stable value (opening new documents in a text editor will change the title of the main window) whereas its help text is less likely to change. The **role** however is a more stable property.

Thus, to identify a GUI state s, we take all widgets $w \in W(s)$ and simply consider a subset ABS_PROP of stable properties from all properties of all widgets on the screen. This subset ABS_PROP defines what we call an *abstraction function* $P_{\text{ABS\_PROP}}(w, s) \subseteq P(w, s)$, that is, $P_{\text{ABS\_PROP}}(w, s) = \{w.p | w \in W(s) \wedge p \in \text{ABS\_PROP}\}$. The abstraction function is configurable in the test settings of TESTAR.

Per default, we take ABS_PROP to be {**role**, **title**, **position**, **enabled**}.

Because this might be a lot of property values to take into account, we will only save a hash value generated from these values. TESTAR recursively calculates a unique hash for each widget, based on the concatenation of the mentioned attributes. It then combines the hashes for the widgets and uses them to calculate the unique hash for the state. Of course this could lead to collisions. However, for the sake of simplicity, we assume that this is unlikely and does not significantly affect the optimization process.

The same approach can be applied to represent actions. Each action type, however, may have parameters. For example, a click action has two parameters: the button (i.e. left or right) and the clicking position (x and y coordinates). Action identifiers need to also take these parameters into account. The method of calculation is as follows: for an action identifier, TESTAR takes the identifier for the state it is currently in and concatenates to that a hash based on details of the action. These details include mouse cursor position and the key that was typed. A unique hash is then calculated over this concatenation. For example, to create a unique identifier for a click on a button, we can use a combination of the button's property values, such as its **role**, **title**, **help text** or its **path** in the widget hierarchy (parents/children/siblings). To create a unique identifier for a text field, if the action identifier takes the entered text into account, then the action that types the text *foo* will have a different identifier than the action that types *boo*.

4.5. Evaluate the new states to find failures (oracles)

A test oracle is a mechanism that distinguishes between a passed or failed test case. In scriptless testing, the test sequence is generated one step at a time during the execution as explained in the previous section. The test oracles check each state we visit. That means that TESTAR oracles define verdicts over states, we call these *online* or *on-the-fly state oracles*. Without specifying anything, TESTAR can detect the violation of general purpose system requirements, or implicit oracles, like those stating that the system should not

- crash, that is, an *unexpected close*;
- freeze, that is, get in an *unresponsive state*; and

DOI: 10.1002/stvr

- contain any *suspicious titles* in any of the GUI widgets.

Suspicious titles can be easily specified using regular expressions. Below is an example.

Example 4: Regular expression for suspicious titles

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

When this oracle is active, in each state *s* that is visited while generating the test sequence, it is checked whether the patterns defined by the regular expression of the suspicious titles appear in the widgets that make up *W*(*s*). A good example from web testing could be defining the HTML error codes in suspicious titles to detect dead links that throw *404 Not Found* error.

TESTAR also allows the user to define more sophisticated application-specific test oracles programmatically in the SUT-specific TESTAR protocol in Java code. Let us look at an example that checks for a security vulnerability. The OWASP[6] lists a vulnerability for *Information exposure through query strings in url*. When sensitive data are passed to parameters in the URL, attackers can easily obtain sensitive data such as usernames, passwords, tokens (authX), database details and any other potentially sensitive data. Simply using HTTPS does not resolve this vulnerability; it should be prevented from appearing in the URL. The following example oracle can detect these vulnerabilities.

Example 5: Programmatic Java oracle

```
1  Set<String> inputTextData = new HashSet<>();
2
3  method executeAction(Action action){
4      if (action.get(Tags.Role).equals("clickTypeInto")){
5           // Save the inputted text into the set inputTextData
6          inputTextData.add(action.get(Tags.Desc));
7      }
8  }
9
10 method getVerdict(State state){
11     for(String dataText : inputTextData){
12         if(state.get(WebTags.Href).contains(dataText)){
13             return new Verdict(Verdict.SEVERITY_WARNING,
14             "Be careful with sensitive information and http GET method");
16 ---------------------------------------------------------------------------
15         }
17     }
18 }
```

In line 1, we define a variable `inputTextData` in which we will store all text that will be entered into textfields while executing actions (lines 3–6) to create test sequences. The oracle (lines 19–14) will check in each state whether elements from `inputTextData` are exposed in the current URL of the SUT.

Besides the online state oracles explained earlier, TESTAR can also interact with the process of desktop applications, listening to the buffers of its process in the *System output* and *Error output* of the operating system. This enables the tester to also define buffer oracles enable to find *suspicious output* coming from the processes, similar in the way that it checks *suspicious titles*. Moreover, we store the output of the processes in logs for later offline manual inspection to find anomalies.

[6]https://www.owasp.org

DOI: 10.1002/stvr

4.6. Runtime execution and modes

The entry point of the TESTAR Java runtime process is the `Main` class. This class has access to the `test.settings` configurations file, defined by the tester. Besides settings like `number_of_sequences`, `number_of_actions` and `SuspiciousTitles`, the tester can define his or her own specific TESTAR **protocol** class that needs to be used for testing. This can be specific for a SUT, a kind of test or just to the tester. A TESTAR protocol is a Java class that is responsible for executing the different parts of test sequence loop as depicted in Figure 1. The code in the protocol class gets compiled at runtime. The SUT-specific, test-specific or tester-specific TESTAR protocols are at the bottom of an inheritance tree as shown in Figure 6.

The `DefaultProtocol` class is the class that contains all the code that actually executes the test sequences. It implements the interface as defined in the `AbstractProtocol` class that contains methods for executing the different parts of test sequence loop (conform Figure 1 and the previous four sections):

- `getState()` (from Section 4.1);
- `deriveActions()` (from Section 4.2);
- `selectAction()` and `executeAction()` (from Section 4.3); and
- `getVerdict()` (from Section 4.5).

The `Desktop` and the `WebdriverProtocol` add a default implementation for specific platforms. Action filtering as it will be explained in Section 6.2 is done by the `ClickFilterLayerProtocol`class. Finally, there is the `RuntimeControlsProtocol` class, which offers controls that allow for the manipulation of TESTAR's runtime modes during execution. There are currently four modes of runtime execution:

- The SPY mode can be used to inspect the widgets of the SUT and see all the information that TESTAR is able to extract. In this mode, actions can be filtered (Section 6.2).
- In the GENERATE mode, the test cycle depicted in Figure 1 is executed.
- The RECORD mode can be used to manually interact with the SUT and store the actions into test sequences.
- The REPLAY mode permits replaying an existing test sequence.

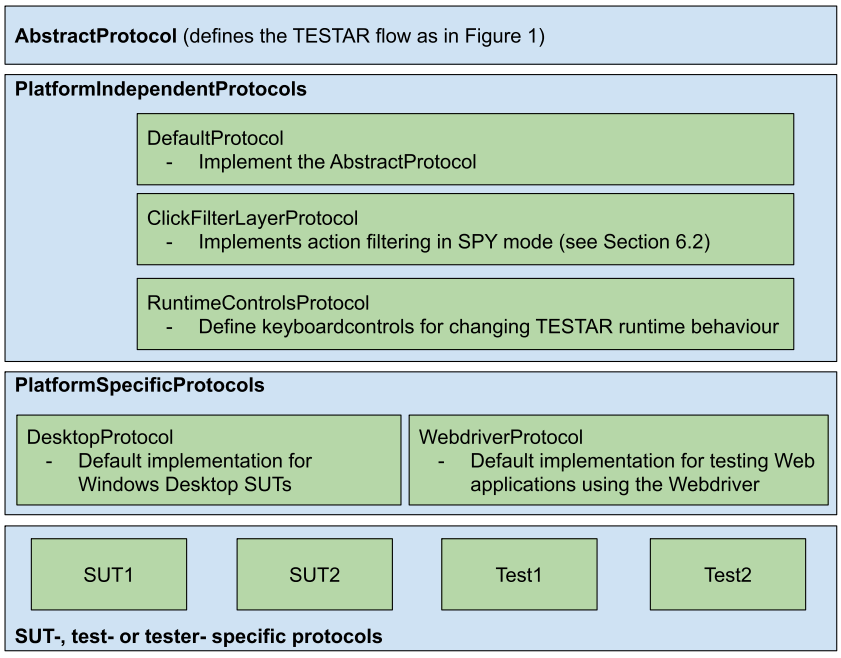

Figure 6. Layers of the different TESTAR protocols. SUT, system under test.

DOI: 10.1002/stvr

4.7. Test results

As explained in the previous sections, a TESTAR run results in a specified number of test sequences with a specified number of actions that have been executed. For each of the resulting sequences, the following information is saved in a directory with a name composed of a timestamp and the name of the SUT:

- Logs that include all the executed actions together with the target widget and the different states of the test sequence, as well as a timestamp that can help synchronize results with other applications.
- Screenshot images that capture the GUI state after each action in a sequence. For this, the coordinates of the states and widgets obtained through the API are used.
- HTML reports to help users follow the flow of executed actions; they combine the textual information of the API and the visual screenshots to display the different sequences.
- Sequences replayable by TESTAR in the REPLAY mode (`.testar` format). These sequences are classified in directories according to the final verdict obtained from the defined oracles (i.e. *unexpected close*, *unresponsive* and *suspicious titles*). These sequences consist of a java object stream that saves the object information of states, actions and widgets.

All the results of a TESTAR run are saved in a directory with a name composed of a timestamp and the name of the SUT. An index log is created during the first TESTAR run and is updated with each sequence execution. This index is useful when we need to support the integration and synchronization of TESTAR with other applications. We then simply use the timestamps and search for all TESTAR sequences by following the path of the right timestamped directory (Figure 7).

4.8. Comparing TESTAR to other similar tools

This chapter compares TESTAR with other scriptless GUI testing tools and highlights the main differences between them. The tools for the comparison were selected based on following criterion: the tool has to be a smart monkey testing tool, based on dynamic analysis during automated exploration of the GUI. Purely mobile testing tools were not included, because TESTAR does not support mobile testing yet so cannot really be compared with those.

The comparison can be found in Table 2 where we summarize the implementation language, license, types of SUTs that can be tested with the tools, how the actions are selected while creating the test sequences, which models are used or inferred during the testing, which oracles are being used to detect failures and which actions are being considered as components for the sequences.

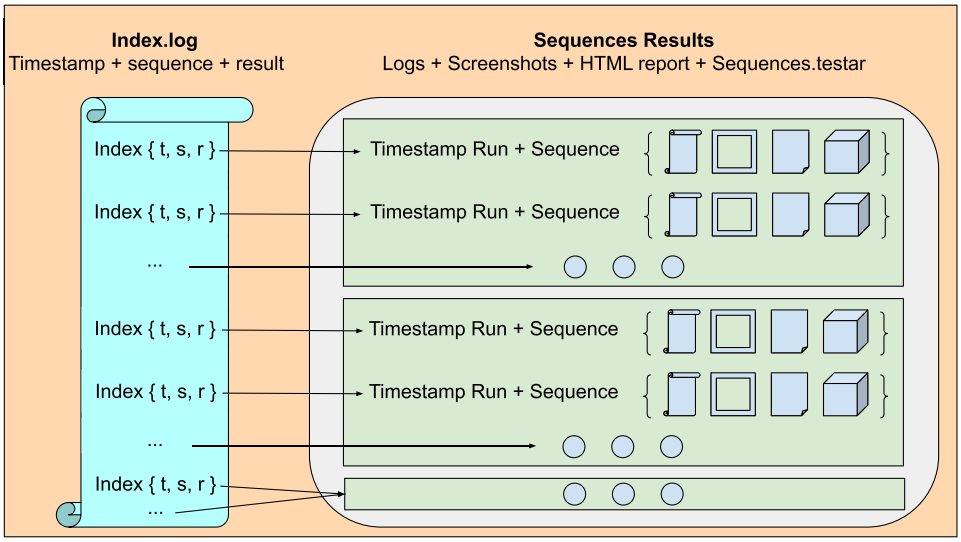

Figure 7. Output structure for test results.

DOI: 10.1002/stvr

Table 2. The state of a GUI as a widget tree.

	TESTAR	Murphy [67]	GUI Driver [43]	Crawljax/ ATUSA [44, 71]	Webmate [72]	GUITAR [73]	AUGUSTO[74]	AutoBlackTest [MPRS11118]
Impl.	Java	Python	Java	Java	NA Java	Java	Java.NET	
License	OS	OS	NA	OS	Commercial	OS	OS (but depends on IBM Functional tester)	OS (but depends on IBM Functional tester)
SUT types	Windows, Web, Java (AWT, SWT, Swing, FX)	Windows	Java (AWT, SWT, Swing)	Web	Web	Java JFC, Java SWT, Web, UNO (Open Office)	Java (AWT, SWT, Swing)	Java (AWT, SWT, Swing), .NET, Windows, Linux
GUI libraries	UIAutomation, WebDriver, Java Access Bridge	UIAutomation, image recognition	Jemmy Java library	WebDriver	WebDriver	Java Accessibility, WebDriver, UNO Accessibility	IBM Functional Tester	IBM Functional Tester, Selenium.
Action selection	Random, programmable, model based, RL, evolutionary algorithms	Random, state model based, programmable triggers	Random, state model based, user actions captured into model for specific inputs	K shortest paths algorithm	inferring state model and using it for crawling, 1Dijkstra's shortest path	Graph model based, configurable triggers,	Depth first and model based	Random, RL, State model
Model inference	State graphs, configurable selection of widget properties define state abstraction	State graphs, data values abstracted from states	State graphs, available actions define state abstraction, data in transitions.	State-flow graph of the DOM states and event based transitions between them	Finite state automaton, multiple options for state abstraction	GUI event flow graph (GUI event extracted from widget properties)	GUI event flow graph. Nodes represent GUI states based on widgets properties.	GUI event flow graph. Nodes represent GUI states based on widgets properties.
Oracles	Crashes, freezes, suspicious elements in widget properties and system outputs, programmable.	Crashes, freezes, programmable	Crashes, freezes, exceptions and errors in system outputs	client/server side exceptions and errors, Dead Clickables, Inconsistent Back-Button	Cross-browser differences, crashes, freezes, generic HTTP and JavaScript errors, programmable	Crashes, State Verifier (model workflow), programmable	Functional oracles that can reveal non-crashing faults	Crashes, Hangs, Uncaught Exceptions, violation of Assertions
GUI actions	Left/right mouse clicks, double	Mouse click actions and		only click actions	Mouse click actions, hovering,		Mouse clicks, text input,	Mouse clicks, text input,

(*Continues*)

Table 2. (Continued)

TESTAR	Murphy [67]	GUI Driver [43]	Crawljax/ ATUSA [44, 71]	Webmate [72]	GUITAR [73]	AUGUSTO[74]	AutoBlackTest [MPRS11118]
clicks, text input, drag-and drop, keyboard events.	text input actions	Mouse click actions and text input actions		heuristics for textual input generation	Invoke GUI APIlibrary event methods	selection, complex actions (fill-in forms interact with color selector, etc.)	selection, complex actions (fill-in forms interact with color selector, etc.)

DOI: 10.1002/stvr

We can see that Java is the most prevalent language used for implementation and that, for now, the majority of all tools are open source. Most of the tools implement different stochastic action selection algorithms; only few depend on deterministic approaches. Models used or inferred are basically state graphs or event flow graphs, and most tools rely on implicit oracles.

5. ADDING MEMORY TO THE MONKEY: STATE MODELS

In the previous sections, we have described how the current state of the GUI can be obtained in a form of a widget tree. This section describes how all the widget trees can be used for extracting or inferring a state model during the automated GUI exploration.

When a model is learned or inferred based on observing a system during its execution, the model includes only the parts that have been visited during the executions. Therefore, an important part of the process is reaching all parts of the GUI. Moreover, if the system produces wrong outputs or other unwanted behaviour during the execution, these errors will also be included in the model. Consequently, the inferred models are not representing the expected behaviour of the SUT, usually captured in requirements and manually defined test oracles. Depending on how the models are used, other means for validating the correctness of the models might be needed. However, an inferred state model can still be used in various interesting ways:

- With a suitable visualization, the behaviour of the system can be analysed through the models and used by an expert, for example, as a base for conformance testing. The visualization is described in Section 5.2.
- When models are created for different versions of the same SUT, the difference between the versions can be detected by comparing the models [67]. This functionality is currently being implemented into TESTAR and is discussed in Section 5.3.
- The model could be used as data for calculating test adequacy indicators like GUI coverage metrics. This will be discussed in Section 5.4.
- The model can be used for a new type of oracles, i.e. offline test oracles [76]. This is discussed in Section 5.5
- The model can be used as input for smart action selection algorithms, for example, based on artificial intelligence, machine learning or search-based optimization techniques. This will be discussed in Section 6.4.4.

5.1. Active learning of state models

Model inference during GUI testing with TESTAR is like drawing a map to remember where have you been and what you have done. The TESTAR model inference approach can be classified as dynamic analysis, as the model is based on observing the system during its automated execution.

In earlier versions of TESTAR, the state model was saved into a file and kept in memory during TESTAR execution. The current TESTAR version, if the state model inference is enabled, will save the model into a graph database, using OrientDB[7] by default. The models are defined in terms of vertices and edges that allow for using graph theory, for example, to navigate the graph, while also having each vertex and edge be a document in its own right, allowing for the required storage of the SUT data in the document's attributes.

TESTAR uses the information of the inferred widget trees to build the state model in three layers:

1 The top layer is an abstract state model of the SUT. It is built incrementally based on multiple test runs. If the SUT name and SUT version of the test runs are the same, then they are used for building the same abstract state model.
2 The middle layer of the model is a concrete state model of the SUT. It is also built incrementally based on multiple test runs, but it will in general contain a lot more state nodes. We will explain the clustering mechanics of these two layers shortly. Each state of the concrete layer is connected to one of the states of the abstract layer, and each concrete action connected to one of the abstract transitions or actions. It is important to note that while the model is identified in

[7]https://orientdb.com/

DOI: 10.1002/stvr

the abstract layer, the other two layers are attached to it. This means that a change in application name or version will not only automatically change the abstract layer of the model but also the other two layers. It is therefore not possible that a single concrete layer can be attached to the abstract layers of two different models.

3 The bottom layer is the management layer, recording meta information about the executed test sequences. Where the abstract and concrete layers describe the SUT, the management layer describes the TESTAR executions.The individual test executions of the management layer are connected to the state nodes of the concrete layer, and a state model of a specific version of a specific SUT is connected to the corresponding model of the abstract layer.

Because the state model implementation uses OrientDB's graph database capabilities, each of our stored entities is either a node or an edge. Nodes in the abstract layer represent an abstract state and are connected by edges that represent abstract actions. Those in the concrete layer represent concrete states and are connected by edges that represent concrete actions. The terms 'abstract state' and 'concrete state' signify nothing more than the identifier that is calculated by TESTAR for a

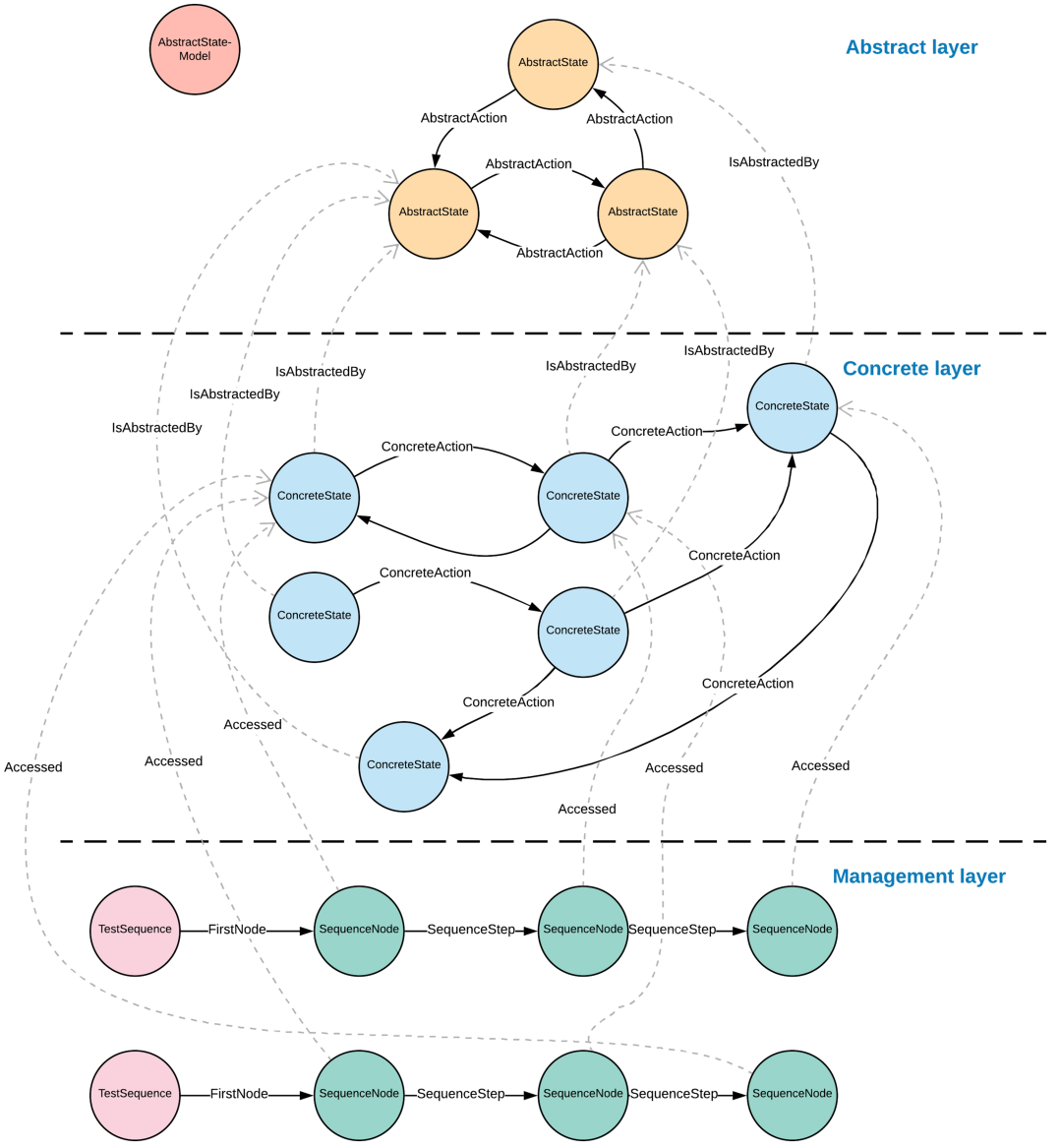

Figure 8. Example of a generated three-layer model in OrientDB.

DOI: 10.1002/stvr

given visual state in the SUT. In Section 4.4, we explained that TESTAR allows the tester to configure which properties of the widgets in the widget tree are used for abstraction, from a selection of around 170 properties. When all the properties are used, we obtain a concrete state identifier. When a selected subset of the properties is used, we obtain the abstract state identifier.

The only clustering heuristics that are used in the top two layers of the state model are these identifiers. We use the abstract state identifier to obtain a node in the abstract layer. If two SUT widget trees produce the same abstract state identifier, they are said to represent the same abstract state node in the abstract layer. The same applies for the concrete state identifier and the concrete layer, and for the abstract and concrete action identifiers. It is easy to see that because the concrete state identifier will almost always use a lot more widget attributes in its calculation, there will be a lot more unique concrete states than abstract states. Because the concrete states are as unique as we can make them, we use the concrete layer for storing information about the application in each visual state, like creating and saving a screenshot. This would not be possible in the abstract layer, because a state in that layer could represent multiple visual states in the SUT, which can have entirely different screenshots. In the concrete layer, this is much less likely to happen.

The management layer is the odd duck out in the model, because no joint model is being built in this layer. Every time TESTAR starts a new sequence within a test run, a new test sequence node is created in the model. The identifier for this node is simply a number that starts with 1 and will increment with each consecutive sequence that is started. It will contain meta information about the run, such as the date and time on which it was started and whether or not the run terminated successfully or was cancelled prematurely due to an encountered error. Every time an action is executed, meta information about the execution of the action is stored in a sequence step edge in the management layer. And when the action has completed and a (new) state is accessed, meta information about the accessing of that state is stored in a sequence node. An example is whether or not an error was occurred while accessing the state. While the graph database does not perfectly lend itself to the purpose of storing meta information in this fashion, by using the right graph queries, a

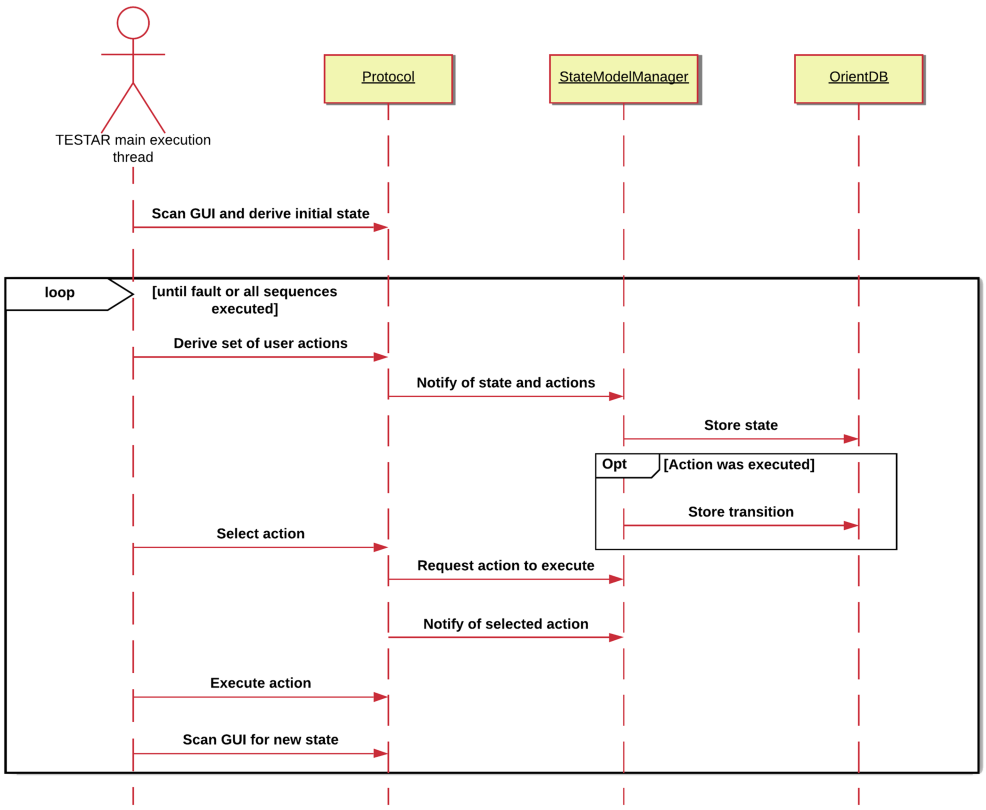

Figure 9. Building the state model in the graph database. GUI, graphical user interface.

DOI: 10.1002/stvr

perfectly usable management overview can be displayed showing the overall progress and success rates of the tests. It also means only one database store can be used to contain all three layers.

Figure 8 shows an example of a generated three-layer model, indicating how the three layers are interconnected. It shows how a state in the abstract layer can be the abstraction for one or more states in the concrete layer and how they in turn have been accessed by a node in the management layer, representing a visitation of these states during a test run. Every TestSequence node represents a tree, in which each node only ever has one child node. When the test sequence ends, the tree is finished and a new tree is started (when a new test sequence begins).

Not shown in Figure 8 is how, during the model inference, when TESTAR finds a new, previously unencountered state, it connects all the abstract transitions of the actions that have not been executed yet to a 'black hole' node, meaning that the behaviour of those actions are unknown. The reason for this is purely technical, as the graph database requires all transitions to have a starting node and ending node.

While all three layers are stored in the OrientDB data store, TESTAR also keeps an in-memory copy of the abstract layer. Doing so allows for fast lookup of information about the states and actions in this layer, which is needed when the model is used as a driver for TESTAR's action selection. Access to the state model functionality is provided by a StateModelManager class, which functions as a gatekeeper. TESTAR provides it with states and actions, and it transforms them into entities that can be kept in memory, stored in the OrientDB data store or both.

Figure 9 shows how TESTAR stores the various properties in the graph database when the execution flow presented in Figure 1 is executed. TESTAR execution can then be explained through the following steps:

1 After starting the SUT, its initial state is captured by the tool.
2 Next, the tool analyses the possible actions it can execute in the current state.
3 Both the state and the list of executable actions are provided to the `StateModelManager`class. If this is the initial state, just the state is stored. Otherwise, a transition between the state and the previous state is also created and recorded. As we explained earlier in this section, a state gets saved in both the abstract and concrete layers of the state model.
4 A copy of the abstract version of the state is also added to the in-memory state model, kept by the `StateModelManager`.
5 The tool asks the `StateModelManager` for an action to execute. It applies one of several custom state selection algorithms to its in-memory state model and returns an action.
6 The tool has the ability to overrule the action that was provided by the `StateModelManager`, so it notifies it of the actual action that was chosen for execution.
7 The chosen action is executed, and the tool waits until another state in the application is reached.
8 The next state is reached. This can either be a previously visited state or a new state. From here, steps 2–7 are repeated and the model is automatically created.

An important part of inferring these abstract state models is selecting a suitable level of abstraction. If the level of abstraction is too low, the resulting abstract layer of the state model might have too many states to be practically useful. However, if the level of abstraction is too high, the model might become non-deterministic and therefore unpredictable, for example, when used for driving the action selection.

5.2. Visualization

Although it is possible to query the OrientDB data store for quite a bit of information about the inferred state models and output it in, for example, tabular format, there are other ways to visualize the data that are more intuitively understandable for humans. Ideally, the visualization should match with a user's idea of what a state model should look like. Because the state model is stored as a graph and a graph is usually the visualization of choice for state models, a graph visualization makes for a great choice.

A state model visualization would preferably serve several use cases, such as the following:

DOI: 10.1002/stvr

- It can provide an overall view of the entire application state.
- It can provide detail information about a certain state or action.
- It would provide a view of the execution paths that lead to the creation of the state model.
- It would provide metrics of different sorts about the state model and its creation.
- It would provide easy insight into states where errors were generated and the execution path that can reproduce it.

There are several open-source graph visualization tools available, such as Gephi [77], GraphViz [78] and Cytoscape [79], which would satisfy some of these requirements. The process of getting the data out of the OrientDB data store and into one of those tools can be quite tedious and elaborate however, cutting down on an intuitive user experience. For that reason, and to support all of the listed use cases, TESTAR comes with a custom state model visualization.

Figure 10 shows a listing of generated models and the test runs that were executed to create them. The listing shows whether the test run was executed successfully ('thumbs up' icon) and whether it was deterministic (i.e. an executed action always ended up in the same state). It is possible to obtain a sequential screenshot visualization of a particular test run by clicking the 'eye' icon. The visualization is a slide show showing the states accessed in the rest run. The column 'Quirks' indicates whether strange results were encountered in model inference during the test run. These could be caused by unsuitable configuration of the level of abstraction used for model inference, for example, causing non-deterministic behaviour.

Besides looking at individual test sequences, it is also possible to look at a graph visualization of the entire state model clicking one of the 'layered' icons. The three layers of the model are represented. Figure 11 shows the display area of the interactive graph viewer. It is possible to interact with the graph and zoom in/out, as well as drag the different graph elements around. Besides these elementary controls, the visualizer comes with a wide array of options to extract information from the generated models. Some of these include the following:

- It is possible to open a side panel for each node in the graph that shows all the information that was stored for that particular state. The information items can be filtered for more easy access.
- For each concrete state, a screenshot can be inspected in full-screen mode.
- It is possible to inspect the widget tree belonging to a certain concrete state, thereby offering great detail on all the widgets that make up the state.
- States containing errors are highlighted for quick lookup.
- From the side panel on each state node, it is possible to create a path trace, showing the exact test sequence(s) that was used to access the state.

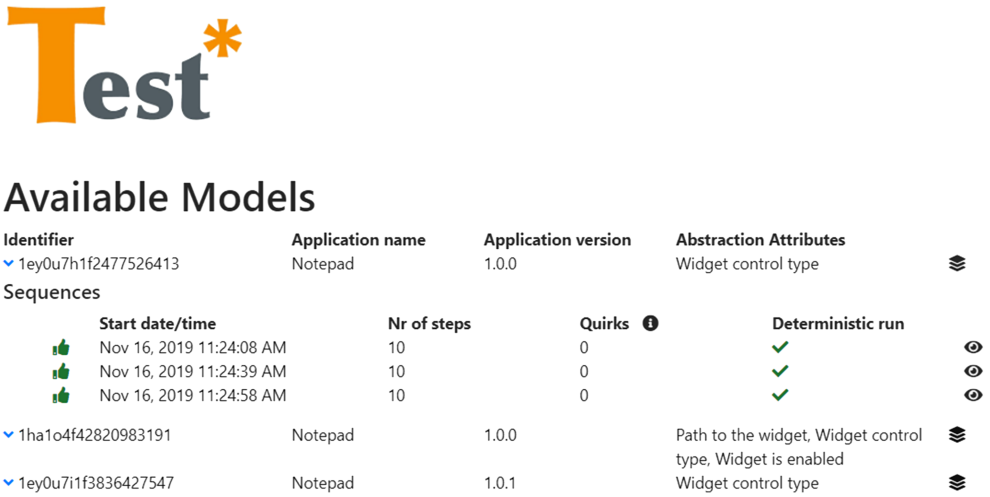

Figure 10. Overview of the test sequences executed for each state model.

DOI: 10.1002/stvr

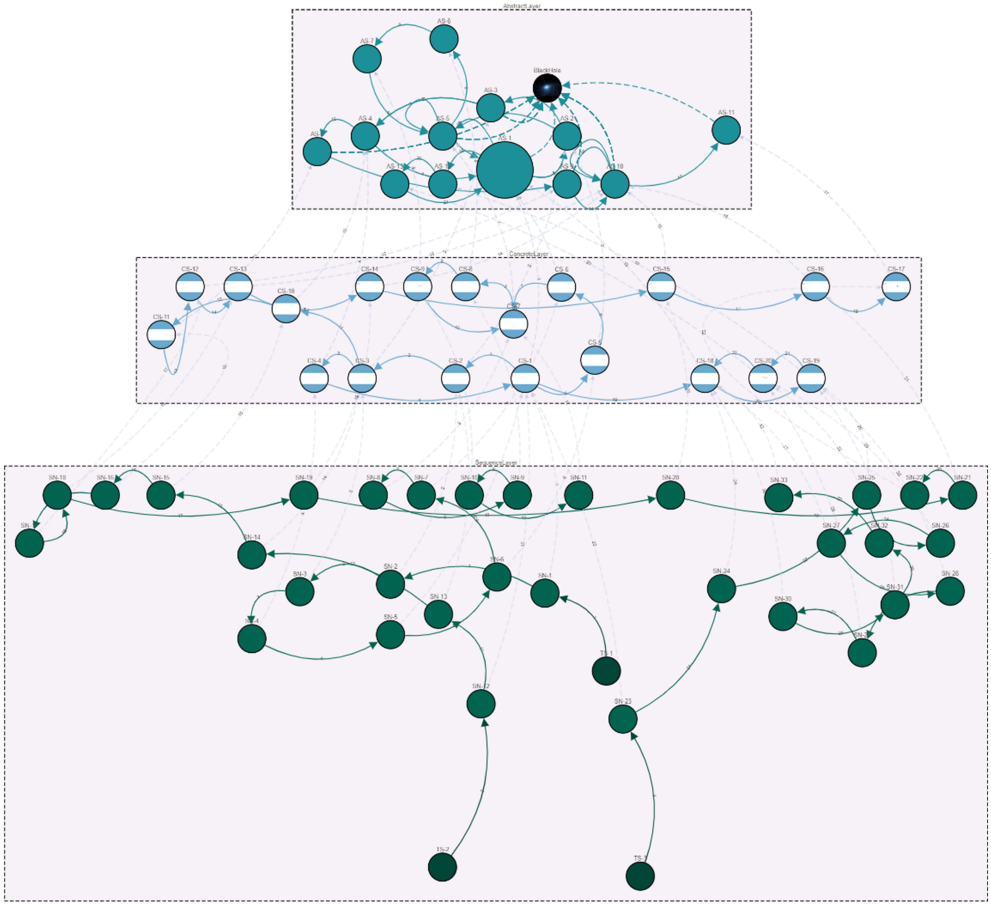

Figure 11. Graph visualization of the state model.

It is evident that the custom visualization provides quite a few options for understanding the data. A downside, however, is that the visualized graph can grow too big and therefore becomes hard to understand. A good candidate for future improvement and research then is to allow the visualization to be chopped into more manageable pieces.

5.3. *Model comparison for automated change detection*

Because learning the state models can be fully automated and executed as part of CI process, it is possible to learn new models automatically from the latest SUT version, for example, once per day. Having the inferred models of the implemented behaviour, it is possible to compare models of two subsequent SUT versions to detect changes between the versions. Of course, without further information, the model comparison will detect both intended and unintended changes.

TESTAR has a proof-of-concept implementation for model comparison but that has not been validated in a case study yet. However, Aho *et al*. [67] report that Murphy tools[8] support model comparison to detect changes between SUT versions, and the feature has been evaluated at F-Secure Ltd.

In the reported evaluation [67], a new model of the latest SUT version was inferred three times per day, and the detected changes were reported by email to the test engineers in form of a link to a web page that visualized the changes with screenshots. The test engineers found the approach useful.

[8]https://github.com/F-Secure/murphy

DOI: 10.1002/stvr

In the reported case study, the data values and other frequently changing parts of the GUI were masked from the comparison, and GUI did not change that much during a day, so the test engineers did not think that analysing the reported changes was taking too much effort. If the change was intended, a part of the normal SUT evolution, it was ignored. If the change was unintended, it was analysed and reported as a bug. Although the results of the evaluation seemed to be very positive, at least the open-source project of the Murphy tools has not been maintained in the recent years.

5.4. Test adequacy indicators

In order to be able to assess the effectiveness of testing with TESTAR and compare different action selection mechanisms, we need to be able to determine what constitutes good or better testing. We need to have appropriate *test adequacy indicators*. Several indicators have been proposed and researched for GUI testing. Memon *et al*. proposed criteria based on the event flow graphs (e.g. coverage of events, event interactions, length *n* event sequences, invocations and invocation terminations) [80]. In [81], it is shown that two factors have a significant effect in the effectiveness of a test suite: the diversity of reached states and the coverage of events. In [82], it is studied what the failure finding capabilities are of test suites looking at the length of the test cases in the suite, the size (number of actions), the length 2 event sequences coverage and the length 3 event sequences coverage. It is concluded that these criteria influence the effectiveness of testing, but more research is needed to find out how they correlate to each other and the failures. In [83], it is described how longer tests perform better than shorter ones in terms of coverage, fault finding and cost. But that the effect eventually diminishes with greater increase in length. McMaster and Memon [84] propose a call stack coverage criterion for stack-based architectures, for which it is shown that larger call trees take more account of the SUT context during execution than code coverage criteria. In [85], t-way interaction coverage criteria are researched for GUI testing. In [86], test adequacy is measured by covering all the event handlers and the data interactions between them. In [87], coverage of all the possible GUI interactions and the meaningful combinations of interactions is defined and measured. In [88], GUI-based mutation operators are defined and empirically evaluated for their usefulness of measuring test effectiveness. In [89], widget coverage is used, but it is concluded that this metric only makes sense when reliable identification of GUI widgets is implemented. Other still use classic code coverage [DBNZ2019] to evaluate test suite quality.

The idea of scriptless test automation is to let the tool automatically execute the system, without the tester monitoring the test execution. The test sequences that are generated during this execution constitute the inferred state models. Consequently, the models can be used for reporting the test coverage only in a way that states where has the tool been and which actions of the modelled area of the GUI have been executed. Because the model only covers the area that has been visited during TESTAR execution, these metrics cannot be used for absolute measurement, but the coverage of individual test sequences can be compared or the combined coverage of all test runs on a specific SUT version can be compared with the coverage of another version. In the context of TESTAR, the following metrics have been used in different studies [90-92]:

- *model states*, the number of unique states in the state transition model;
- *model actions*, the number of unique actions that have been executed, shown as transitions in the model;
- *test actions*, the total number of non-unique actions that have been executed;
- *abstract states*, the number of unique abstract states visited, in which an abstract state is a group of states that are the same when a certain set of properties is disregarded (e.g. the colour of controls);
- *longest path*, the length of the longest path in the state transition model; and
- *state coverage*, the minimum and maximum state coverage, defined as the lowest and highest rate of executed actions over the total number of available actions of a state.

5.5. Offline oracles

The state model can be leveraged to define reusable and *offline oracles* as in [76]. As opposed to the on-the-fly, or online, oracles that perform their evaluation on the current state of the SUT

DOI: 10.1002/stvr

during testing, offline oracles perform their evaluation on stored test data asynchronously *after*testing.

Offline oracles are defined as queries to the graph database and can hence check properties about (combined) test sequences. OrientDB provides multiple ways to query the database: graphs can be traversed using a Structured Query Language dialect, or a Java API, or the Gremlin language[9][93]. Let us look at an example from [76] of a Gremlin queries to define an offline oracle for a state model stored in the OrientDB database. In [76], reusable offline oracles are being studied to test for the accessibility using the guidelines from WCAG2ICT [94]. For example, Guideline 2.4 (Navigable) states that a SUT should provide ways to help users navigate, find content and determine where they are. This guideline can be partially covered by an offline oracle that tests for ambiguous window titles, that is, a check that not too many windows have the same title. The Gremlin implementation in Example 6 assumes that we have the graph database $G = (V, E)$ containing the state model. The query selects all vertices that are widgets (line 2) and that have the WCAG2IsWindow property set (line 3). These widgets are windows. The widgets are grouped (line 4) by their Title property.

Example 6: Obtaining property values

```
1  List<Object> titleCounts =
2    G.V.has('@class','Widget')
3       .has('WCAG2Tags.WCAG2IsWindow.name()',true)
4       .groupCount.by('Tags.Title.name()')
5       .cap
```

The result of executing this query in Example 6 is a map of titles and the number of times they appear in the graph database. Then, to check the oracle, we need to loop through this list and raise count above a predefined allowed threshold as a warning because it means that titles are duplicated and hence ambiguous violating Guideline 2.4.

6. ADDING INTELLIGENCE TO THE MONKEY

As mentioned in Section 3, there are various ways to make a test monkey more intelligent. This section describes the various ways to make TESTAR smarter, knowing more about how to test a specific SUT.

6.1. Advanced derive actions

As explained in Section 4.2, having obtained the GUI's current state, we can go on to derive a set of actions from which we will select one. One thing to keep in mind is that the more actions we can choose from, the bigger the sequence space will be and the more time the search for crashes might consume.

Ideally, we should only select from a small set of actions that are likely to expose faults. Thus, the challenge is to keep the search space as small as possible and as large as necessary for finding faults.

6.1.1. Deriving sensible actions. This strategy means that, for each state, we strive to generate a set of *sensible* actions that should be appropriate to the widgets that they are executed on: buttons should be clicked, scroll bars should be dragged and text boxes should be filled with text. Furthermore, we would like to exercise only those widgets that are *enabled* and not *blocked*. For example, in a window that is blocked by a message box, it would not make sense to click on any widget behind the message box. Because the box blocks the input, it is unlikely that any event handling code (with potential faults) will be invoked. Putting more intelligence into derive actions will reduce the choices for uninteresting actions during action selection.

[9]https://tinkerpop.apache.org

DOI: 10.1002/stvr

6.1.2. Deriving top level actions. Widget that are on top of the layout hierarchy are more likely to lead to actions that trigger state transitions. Elements like Menus or emerging Windows existing on the SUTs usually contain and are designed to include a functional flow of the application. In order to favour top actions, TESTAR implements a prioritization approach based on an internally defined property called *z-index*. The *z*-index of widgets presents the position in the stack of windows. The window with the highest *z*-index is on top. This gives the possibility to derive the actions from top level widgets.

6.1.3. Deriving new actions. Another prioritization approach for a faster GUI exploration is comparing the available actions of the current state and the previous state to detect which of the currently available actions are new. In case some action requires more than one step to execute, like first opening the file menu and then choosing and menu item, such a prioritization would increase the chances of triggering the new actions opened after clicking the file menu.

6.2. Filter actions

Besides telling TESTAR the actions that it can do, it is also important to tell what it should *not* do. Action filters can be defined for this purpose. Letting TESTAR randomly interact with widgets on a GUI could trigger 'hazardous' operations like delete or overwrite files, possibly damaging the operating system. A way to safeguard this is by simply running test monkeys on safe environment, like a virtual machine that can be easily recovered or a sandbox that you can break. Still, filtering actionable widgets is useful to make sure we concentrate on actions that lead to testing the SUT. For example, actions that minimize/maximize a window, close the SUT or open a help menu that simply goes to a web site outside the SUT are not really interesting for testing. Filtering those will reduce the search space and save time during the test execution.

The action filters can also be used for defining specific areas for scriptless testing, for example, filtering different menu options or force the tool to test a specific area of the SUT. Action filtering can be done in three different ways with TESTAR, and we will discuss each of them next.

6.2.1. Filtering with regular expressions. In a way similar to the use of regular expressions defining oracles for detecting suspicious titles (Example 4), we can also use regular expressions to filter those widgets whose title matches a regular expression. For example:

Example 7: Regular expression to filter widgets

```
WidgetTitleFilter = .*[cC]lose.*
                  |.*[mM]inimi[zs]e.*
                  |.*[sS]ave.*
                  |.*[pP]rint.*
```

When this filter is active, in each state s that is visited while generating the test sequence, the title property w.`title` of all widgets $w \in W(s)$ is matched against the regular expression. In case of a match, actions on that widget will not be considered. The `WidgetTitleFilter` from above is a general purpose filter pattern that is useful for almost all SUTs, because we do not want to close or minimize the application we are testing, we do not want to allow files to be saved causing hazardous actions and we do not want to access to the print menu of the system for printing documents.

6.2.2. Using the click filter. TESTAR click filter functionality allows testers to filter widgets just by clicking on them through the GUI of the SUT during the SPY mode. The filtered widgets are saved into a blacklist, which means that they are not actionable anymore and actions will not be derived for them. We can undo filtering widgets in the same way, even if the filtering was achieved by matching a regular expression.

To make filtering work accurately, we rely on the uniqueness of the abstract identifiers explained in Section 4.2. It is important to select the right level of abstraction and precision to guarantee

DOI: 10.1002/stvr

uniqueness. For example, consider an OK button. If we would use the **role** (i.e. button) and **title** (i.e. OK) properties for the abstract identifier, it will filter all OK buttons in all the different states of the SUT. Adding the **path** property would make the filtering more effective, because the path of the button in the widget tree will most likely be different in different states.

6.2.3. Programmatic filtering. The third and most flexible way to filter actions is programming the desired behaviour in the SUT-specific, test-specific or tester-specific TESTAR Java protocol.

In some SUTs, the configured set of properties may not be sufficient for proper filtering of the existing widgets, for example, because they use a different set of accessibility properties. Then programmatic filtering is the best option.

As TESTAR offers the possibility to customize all the methods of its execution flow (see Figure 1), users can prepare a specific filtering of widget actions using the properties that they consider appropriate. If we take a look at Figure 3, we can see that properties such as **href**, **helpText** or **class** can be useful for filtering widgets. In *deriveActions()* function of some specific TESTAR protocol, we can filter the actions with the conditions we need. In Example 8, all widgets that have some undesired URL in their **href** tag property are filtered.

Example 8: Filtering widgets programmatically

```
1 for (widget : state)
2     if (widget.get(WebTags.Href).contains("Undesired-URL"))
3         continue; // skip this widget
4     else // derive actions as defined
```

6.3. Triggering pre-specified actions

Quite often, reaching all parts of the GUI requires using a specific input in a specific state of the GUI; for example, a login screen requires a valid username and password. It would not make much sense to try inputting random usernames and passwords into the text fields. Instead, before TESTAR execution, the user should specify the valid input and how to recognize the fields into which the input has to be written. TESTAR allows triggering pre-specified actions when a specific state has been reached programmatically in the SUT-specific, test-specific or tester-specific protocol.

The start-up sequence is a special case of pre-specified actions for TESTAR as there is a specific Java function for defining them in the TESTAR protocol class and this function is executed each time after the SUT has been started. When triggering the actions during the TESTAR execution, it is done by looking for one or several widgets with specified properties to recognize the state when the pre-specified actions should be executed. If a matching state is detected, then the pre-specified actions are executed instead of letting the action selection algorithm to choose. After executing the pre-specified actions, TESTAR returns to the normal flow, driven by the action selection algorithm.

If the SUT has been built with test automation and testability in mind, each input field should have a unique automation ID. Then that ID can be used to recognize both the state and the field where the pre-specified input has to be provided. Otherwise, another, hopefully unique property value has to be used.

A `CompoundAction` (Section 4.2) can be used, if more than one action is required to fulfil the purpose of the pre-specified actions. For example, a login screen would probably require two actions for filling username and password and a third action to click on Submit or OK button. The benefit of the compound actions is that they require recognizing the state of the GUI only once, when it is triggered, and executing multiple actions at once. An example of this is below. In this example, `Type` is a class that represents typing actions, and `KeyDown` is a class that represents pressing the argument key.

DOI: 10.1002/stvr

Example 9: Compound actions for login

```
for (widget : state){
    if (widget.get(Tags.Title).contains("Sign in")){
        new CompoundAction.Builder()
            .add(new Type("username"))
            .add(new KeyDown(KBKeys.VK_TAB))
            .add(new Type("password"))
            .add(new KeyDown(KBKeys.VK_ENTER)).build();
    }
}
```

For the same logging purpose, it is also possible to define a pre-specified CompoundAction using WebDriver actions. Because these actions are internally defined with the specific DOM API web method, the user only has to indicate the matching attribute value to find the web element and the desired value of the attribute he wants to overwrite.

Example 10: Pre-specified actions for login using theWebDriver

```
CompoundAction.Builder builder = new CompoundAction.Builder();
for (widget : state){
    if (widget.get(WebTags.Id).contains("username")){
        builder.add(new WdAttributeAction("username", "key", "value"));
    }
    else if (widget.get(WebTags.Id).contains("password")){
            builder.add(new WdAttributeAction("password", "key", "value"));
    }
}
builder.add(new WdSubmitAction("Form Name")).build();
```

6.4. Advanced action selection

The action selection strategy is a crucial feature for scriptless testing. The right actions can improve the likelihood and decrease the time required for finding failures. As indicated before, the action selection characterizes the fundamental challenge of intelligent systems: *what to do next*.

As we indicated before, in default mode, TESTAR selects the actions randomly. The current state of the TESTAR at this writing consists of work on adding intelligence to the action selection strategy using reinforcement learning, ACO and meta-heuristics. We will summarize these in the following subsections.

6.4.1. Reinforcement learning. In [95], [7] and [90], we have used Q-learning, a reinforcement learning technique, to improve action selection. The objective of Q-learning is to guide an agent in the process of learning what action to take under different circumstances. The agent, at a state s, must choose one among the set of actions $A(s)$ derived at that state. By performing an action $a \in A(s)$, the agent can move from state to state, until it reaches the goal. Executing an action in a specific state provides the agent with a reward (a numerical score that measures the utility of executing a given action in a given state). The goal of the agent is to maximize its total reward by learning which action is optimal for each state. The action that is optimal for each state is the action that has the highest long-term reward.

Using machine learning algorithms for action selection requires a state model that maintains the learned information, like the Q values for the actions in a specific state. The previously mentioned Q-learning studies used an older implementation of TESTAR state models, based on files and in-memory model, instead of the novel inferred state models stored in a graph database introduced

DOI: 10.1002/stvr

in Section 5. Abstraction of the state model has a big impact on the success of using machine learning for the action selection. If the model is too concrete and therefore suffers from state space explosion, the learning algorithm will suffer because it won't get to use the learned values unless it returns to a state that it has visited before. We are currently researching state model abstraction, described in Section 8.2.

The Q-learning algorithm used in [95], [7] and [90] is governed by two parameters: the maximum reward, R_{max}, and the discount γ. Depending on how these are chosen, the algorithm will promote exploration or exploitation of the search space. The R_{max} parameter determines the initial reward for unexplored actions; so a high value biases the search towards executing unexplored actions. On the other hand, discount γ establishes how the reward of an action decreases after being executed. Small γ values decrease the reward faster and vice versa. The reward function $R(s, a, s')$ decreases every time when action a is executed making it less attractive for the agent to select. Note that our reward function is non-Markovian, because it depends on previous state transitions. The policy $\pi : S \rightarrow A$ just selects the action with the highest Q-value.

Experiments with different Q-learning parameters were performed, and they were compared with random action selection. A good selection of the parameters when using Q-learning needed to be emphasized after this study, because the different parameter settings for Q-learning allowed to obtain better and worse results than random action selection.

6.4.1.1. Related and future work

First of all, *different reward functions* should be investigated, and their performance should be measured for the effectiveness of testing. The fact that our reward function is not Markovian might have caused it to behave similar to random for some configurations, although it was observed that more exploration was done. This might be due to the fact that during the learning process, paths are searched systematically in the state space by keeping a memory where it has already been searched this happens to end up in the Q-values. In [AKKB18 96], a similar reward function is used as in our work. Others have used different reward functions in the GUI testing setting; it remains to be investigated how these perform within TESTAR. In [MPRS12 97], higher rewards are given to actions that cause more changes to the states. Consider a state change $s \rightarrow_a s'$, the reward value $R(s, a, s')$ is calculated by counting the number of widgets that are present in s and changed in s'. A widget that disappeared gets reward 1. A widget that is still there in s' adds to the reward with a fraction of the properties (used to identify the abstract state) that changed value. In [DBNZ2019 98], rewards are only given to actions that cause a visual change on the GUI that is determined by comparing the screenshots before and after performing the action. In addition, probabilistic distributions were used to guide exploration and exploitation, based on the probability of obtaining a reward when executing an action on a widget in a given state. In [KS2018 99], matrices of Q-values are learned offline for all applications. This avoids the need for learning the Q-values for each application during testing. The Q-matrix contains five abstract categories of states defined according to the number of enabled actions in the state, and seven abstract types of actions are considered (menu, back, click, long click, test, swipe and contextual). Rewards are given for increased (different) state coverage and crash detection. In [GC17 100], different rewards are given to different types of actions based on the transitions they cause, that is, transition to another activity or a crash get reward 1, and transitions that go to another application get -1; other transitions do not get a reward.

Besides rewards and offline learning, different *policies* should be investigated. And together with the reward functions, optimality criteria should be studied [101] and how these are related to learning the best action to select for testing through the GUI. For all these topics, the adequacy criteria for GUI testing mentioned before, a research topic by itself, are needed.

6.4.2. Ant colony. In [102], ACO is used to optimize McMaster's stack coverage criterion (Section 5.4). Experiments are performed with a tool that can be considered an ancestor of TESTAR. In order to obtain method call trees while executing a SUT, the Java virtual machine was instrumented. Hence, no source code was needed to obtain the metric.

Ant colony optimization seems to be a good candidate for optimizing test sequences because it is a combinatorial optimization technique [103], meaning that it looks for a solution that consists of a

DOI: 10.1002/stvr

combination of unique components selected from a typically finite set. The objective is to find the optimal combination of components (i.e. actions in a test sequence). ACO does not use a mutation operator and hence maintains closure of the solutions; this is needed in our context because mutations to a test sequence could easily result in infeasible executions of our SUT.

Ant colony optimization maintains an entire pool of the so-called ant trails that correspond to our test sequences. Each test generation builds several trails (= test sequences) by selecting components from a set C (in our case, C is the set of GUI actions A). The likelihood of an action being chosen is determined by its pheromone value. When a test sequence is finished, we evaluate its fitness in terms of the reached stack coverage; the pheromones of the actions contained within that trail are updated according to the fitness value. Thus, the pheromones of an action indicate how often it participated in test sequences with high stack coverage. Experiments show that ACO continuously finds test sequences with better coverage than the random strategy.

6.4.2.1. Related and future work

Future work needs to investigate the fault-finding capabilities of the generated sequences. For this, we need applications with known faults. Also, other adequacy criteria can be investigated; this is a must for web applications, where the stack coverage criterion cannot be measured. In [104], ACO is applied to GUI testing where the optimization criterion is similar to [?MPRS12]; that is, high pheromone values are given to actions that cause a lot of change. Finally, different optimization techniques that do not have a mutation operator or one that preserves feasibility of the test sequences need to be researched.

6.4.3. Evolving action selection mechanism. In [91,92], we have used evolutionary computing for evolving action selection strategies. In evolutionary computing, individuals are randomly generated from components and then evolved into hopefully fitter individuals. This is done by evaluating each of the individuals to determine their fitness, selecting some of them and performing mutations on the selected individuals to create new individuals. The fitter individuals have a larger chance of being selected for mutations. The assumption is that the individuals can pass on their fitness to their children, so the selections and mutations should result in an even fitter next generation.

The individuals are trees that represent action selection strategies, as shown in Figure 12. As the tree consists of components from a predefined set, trees can be automatically generated and components can be interchanged. The root of the tree is always an if statement. The branches and leaves of the tree are booleans and action selections, with optional operators.

During the evolutionary computing, a population of 20 individuals is used, with a maximum tree size (the number of nodes in a tree) of 20. The steady-state model is used, which means

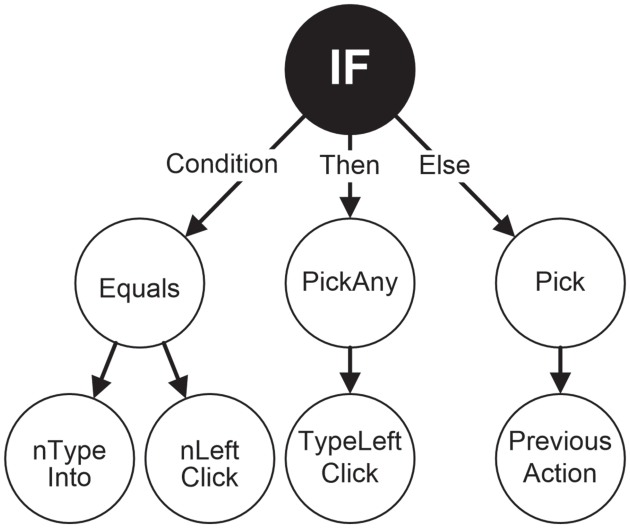

Figure 12. A tree-based representation of a strategy as used in [91,92].

DOI: 10.1002/stvr

that the population is not entirely changed at once, but partly, for example, one individual at a time. Tournament selection of size five is used. This means that five individuals are selected randomly from the total population, and of these five, the fittest individual is picked for further processing. This processing is done using the evolutionary operators, in this case mutation (replacing part of the individual with a randomly generated part) and crossover (exchanging part of the individual with a part of another individual it is crossed with). The evolution is set to end as soon as a strategy is found that results in more than 30 visited states within the configured 100 actions.

The SUT used during the evolution is Microsoft PowerPoint. The strategy found using the evolution is then compared with the random strategy and the Q-learning strategy described in [92]. All three strategies are run on PowerPoint, Odoo (an open-source Enterprise Resource Planning system)[10] and Testona (a commercial application for test case design).[11] The metrics that are used for comparison are the abstract states, the longest path and the minimum and maximum state coverage. Statistical analysis revealed the superiority of this approach (compared with Q-learning and random testing) in some of the SUT used in the experiments.

6.4.3.1. Related and future work

A future line of research will be to use novelty search [105]. Evolutionary algorithms tend to guide the individuals towards peaks of fitness. In novelty search, an individual who is considered new in one generation is probably considered less novel in the next. Novelty search does not reward progress, rather it rewards being different. In the context of TESTAR where the definition of oracles is very complex, algorithms that favour the diversity of solutions could increase the probability of finding a failure.

6.4.4. Prioritization and model-based search for unvisited actions. A state model can be used for an action selection algorithm that plans more than one step ahead. Basically, the state model can be used as a map to check where you are and where you'd like to go, driving the action selection to a specific target. The inferred state model from Section 5.1 can be used for creating an exploratory action selection algorithm that has access to the abstract state model and prioritizes or searches for unvisited actions.

The algorithm will look at the current state in the abstract state model for actions that have not yet been visited by TESTAR. From this collection of unvisited actions, it returns a random one. If there are no unvisited actions left in the current state, it looks at the states that are one action away and checks them for unvisited actions. This process is repeated until an unvisited action is returned or until the entire abstract state model has been covered, in which case a random action is returned from the current abstract state. Such an algorithm is currently being implemented into TESTAR but has not been evaluated for its effectiveness yet.

6.4.4.1. Related and future work

A similar exploratory action selection algorithm has been used in GUI Driver [43] and Murphy tools [67], but its effectiveness has not been compared with other action selection algorithms. In the near future, TESTAR will be used to compare the algorithm with other action selection algorithms, measuring how quickly the algorithms will cover the GUI, possibly looking into code coverage as well.

7. INDUSTRIAL CASE STUDIES

The successful transfer of academic results into industry is important. On the one hand, academic research activities should be guided more towards the challenges of industry and solutions to their immediate problems. On the other hand, practitioners in the industry should help academics in validating their research results within a real industrial context. Technology transfer has always

[10] https://www.odoo.com
[11] https://www.testona.net

DOI: 10.1002/stvr

been on the top priority list of the TESTAR project and remains to be. In this section, we summarize the collaboration projects that have been successfully executed over the courses of the years.

All studies are case driven and have been executed following the Methodological Framework for Evaluating Testing Techniques and Tools (MFEST[3]) described in [106]. The need for this framework emerged during the execution of the EU funded project EvoTest (IST-33472, 2007–2009, [107]) and continued emerging during the EU funded project FITTEST (ICT-257574, 2010–2013, [108]). The framework conforms to the well-known and general guidelines and checklist from case study research [109-112] but has been made specific for the evaluation of software testing treatments.

MFEST[3] distinguishes different scenarios for executing an evaluating study; they increasingly depend on the information that is available for comparing. *Scenario 1* consists of only a qualitative assessment where we do not have enough information to compare. We do not know how many faults there are, we cannot inject errors, we cannot compare with other testing techniques used because they are undocumented, nor do we have a company baseline. However, studying and reporting on the measurements found for effectiveness, efficiency and subjective satisfaction will be done during semi-structured interviews. *Scenario 2* consists of scenario 1, but we can also do some quantitative analysis because we have some sort of company baseline to which we can compare. *Scenario 3* consists of scenario 1 or 2 and a quantitative analysis of fault detection rate because we access to a known set of faults (injected or not). *Scenario 4* consists of scenario 1 or 2 and a quantitative comparison of the tests generated by the approach that is being evaluated and an existing test suite. *Scenario 5* consists of scenario 4 plus the fault detection rate of the different test suites. There are two more scenarios in [106], but we did not do these types of studies with TESTAR yet.

In [106], many metrics are defined to answer research questions related to test effectiveness, efficiency and subjective satisfaction. All the metrics that have been used in the TESTAR studies are numbered in Table 3to make it easier to refer to them in Table 4 that presents a summary of the executed case studies.

In Table 4, the *GUI testing* column describes how GUI testing was done before the case started (M meaning manual and CR meaning using capture and replay). The *scenario* column refers to the scenarios from MFEST[3] described earlier. The *context/subject* column mentions the project in which the study was carried out and indicates how many academics (*aca*) and how many industrialists (*ind*) participated. The numbers in the 'effectiveness, efficiency and subjective satisfaction' columns correspond to those in Table 3.

To try to generalize the results of these case-based studies, we can use an architectural analogy [119]. For this, we need to describe an architecture of the cases, that is, components with interactions, such as the systems, the people and their roles. Except for B&M, who was already doing some automated GUI testing using CR tools, all the cases parted from manual GUI testing practices before the introduction of TESTAR. This can lead to the simple architecture reflected in Figure 13. Within this setting, the studies have shown that TESTAR is an effective complement to existing manual practices and can find undiscovered failures in a SUT without too much costs in setting it up.

Table 3. Metrics from [106] used in the TESTAR studies.

Effectiveness	Efficiency	Subjective satisfaction
1. Number of failures	1. Time needed to design the test suites	1. Reaction cards
2. Code coverage	2. Time needed to run	2. Informal interview
3. Functional test coverage	3. Lines of code for set-up	3. Face questionnaires
4. Number of false positives	4. Time needed for post analysis	
5. Reproducibility		
6. Impact or severity of faults		

DOI: 10.1002/stvr

Table 4. TESTAR case studies

SUT			Evaluation Study			Metrics				Results
Company	Platform	Language/ LOC	GUI testing	Scenario	Context/ Subjects	Publication	Effectiveness	Efficiency	Satisfaction	
Softeam (Large)	Web	PHP 2k	M	5	FITTEST 2 aca 2 ind	[108]	1, 2, 5	1,2	1, 2, 3	The automated tests generated with TESTAR were considered to be able to compete with those of the manual tests of SOFTEAM. The subjects felt confident that if they would invest a bit more time in customizing the action selection and the oracles, the TESTAR tool would do as best or even better as their manual test suite w.r.t. coverage and fault finding capability. This could save them the manual execution of the test suite in the future.
Prodevelop (SME)	Web	Java	M	1	TESTOMAT 1 aca 3 ind	[109]	1, 5	1, 2	2	In order to integrate TESTAR into the existing CI test cycle, a CI architecture needed to be adjusted in which TESTAR could be invoked remotely in a distributed manner. TESTAR during 4 nightly builds for 12 hours with random action selection protocol and a configuration of 30 sequences of 200 actions each night. We found 21 failure sequences that could be traced back to two faults.
Clave (SME)	Windows Desktop	VB	M	1	SHIP 1 aca 2 ind	[110]	1, 5, 4	1, 2, 3	2	Setting up TESTAR took an initial effort of 26 hours and inspection of log files, reproduction and comprehension of errors took around 100 minutes of manual intervention during and after tests. TESTAR detected 10 previously unknown critical faults.
Cap Gemini (Large) and	Web	Java 12k	M	5	OU, UPV 1 aca 3 ind	[111]	1, 3. 6	1, 2	1, 2	Running TESTAR in default mode for 71 hours (192 sequences and 98.081 actions) resulted in finding 4 failures

(*Continues*)

Table 4. (Continued)

SUT			Evaluation Study			Metrics				Results
Company	Platform	Language/ LOC	GUI testing	Scenario	Context/ Subjects	Publication	Effectiveness	Efficiency	Satisfaction	
Prorail (Large)										that the manual test suite did not find and 80% of functional coverage. Of the 4 failures, 1 was not reproducible; the remaining failures were null pointer exception due to clicking more times on button; functional fault while exporting to Excel to release the day plan; concurrent modification exception when clicking two times on a button in a row without waiting. All the failures were assigned high severity since they forced the end user to restore the application for further use. Detailed results of this study will be published in the future.
Kuveyt Türk Bank (Large)	Web	many562k	M	1	TESTOMAT 2 aca 3 ind				2	To restrict TESTAR from testing external Web pages, a SUT-specific protocol class based on the Webdriver was extended with 1) Whitelisted domain URLs that are to be considered part of the SUT, and 2) Blacklisted extensions of resources, such as PDFs. Out-of-domain actions and URLs were prohibited, so that TESTAR did not select external links and returned back to SUT if it ended up to domain outside the SUT. Kuveyt Türk Bank uses Selenium and Appium for regression testing of mobile and internet banking. The objectives for the collaboration is to evaluate script less testing and

(*Continues*)

Table 4. (Continued)

SUT			Evaluation Study				Metrics			Results
Company	Platform	Language/ LOC	GUI testing	Scenario	Context/ Subjects	Publication	Effectiveness	Efficiency	Satisfaction	
										TESTAR tool to see if the maintenance cost can be reduced on the long run, while increasing test coverage and testing the robustness of systems.
Ponsse (Large)	Embedded Windows	VB >1M	M	1	TESTOMAT 1 aca 3 ind	[112]	1, 6	-	2	The solution implemented works as follows. When the Ponsse specific development branch of TESTAR had a new commit, it triggered Ponsse CI to download and build TESTAR and run the defined tests against the latest SUT version. TESTAR found faults that were not discovered with manual GUI testing or script-based test automation.
Indenova (SME)	Web		M	1	SHIP, PERTEST 1 aca 1 ind	[113]	1	1, 2, 3	2	To get TESTAR running, 35 lines of code and 10 minutes were needed. Oracles did not require any lines of code, but just a regular expression with the list of unwanted localised words. Two issues were detected in various places in the SUT during a one hour test session.
B&M (SME)	Windows Desktop	Java 240k	CR	1	FITTEST 1 aca 2 ind	[20]	1	1, 2	2	TESTAR found Null Pointer Exceptions that could be traced to minimizing the main editor. Investigating more, it was found that the tool with minimized main editor was not functionally specified. The company was impressed and convinced that tools like TESTAR can improve the faultfinding effectiveness of current test suites designed at B&M, but only if used complementary to the current practice.

8. FUTURE RESEARCH DIRECTIONS

In this section, we will summarize the ongoing and future research directions that have been put forward in this paper.

8.1. Obtaining the state and test execution

Even if the test execution is automated, testing through GUI is significantly slower than, for example, unit testing. The main reason is that you have to wait for the GUI to update after each executed action. With enough resources, some GUI testing approaches could be executed faster if done in *parallel*. However, model inference would still require a single entry point for collecting the data [4]. In case of TESTAR, that point would be the graph database, but updating the abstract state model to and from multiple simultaneously executed TESTAR instances requires more research.

Another future research directions is supporting scriptless testing in a *remote* way. This requires separating the platform-independent logic from the platform-specific adapter that is able to obtain the state and execute interactions on a specific platform and allowing these two components to communicate remotely. One option would be defining an abstract language to describe all the widgets for the state and all the actions as commands and transformations at the platform-specific adapter. To allow running the adapters inside company firewalls, the adapters should act as clients that register into a server and maintain a connection that allows the server to send commands in a form of responses, for example, using web sockets.

The *cloud* is another relevant future execution environment for GUI testing. The challenges include the need for platform-specific environments, including interactions through mouse and keyboard, and the need to obtain the state of the GUI. For web testing, Selenium WebDriver could provide means to implement support for cloud environments, for example, using containers. For Windows desktop applications, however, this might be more challenging because containers aim to offer micro services, having a minimal set of functionality offered in each container.

Currently, TESTAR starts the WebDriver locally and is therefore able to use the GUI of the browser to interact with the web application using mouse and keyboard actions. In future, a remote connection to Selenium WebDriver should be supported and allow running it in headless mode. The support for *headless mode* could enable running TESTAR in a container, such as Docker, and make it easier to run TESTAR in cloud environment.

Another useful extension would be to support the integration of *Selenium scripts* as part of TESTAR execution. That way, the user could define the triggered behaviour (explained in Section 6.3) as Selenium scripts that in many cases might already exist for scripted functional testing.

Evidently, adding support to test *mobile* applications with TESTAR is a desirable future direction, for example, using Appium [68] to obtain the state of mobile applications.

Another direction for obtaining the state of the GUI would be using *image recognition* on the screenshots of the GUI. The main benefit of this approach would be that it is platform independent, as long as the screenshots are available. There are open-source tools and libraries, such as OpenCV[12] (Open Source Computer Vision Library), that could be used to recognize the widgets of the GUI. Machine learning could be used to train image recognition algorithms for better accuracy and possibly recognizing also the type of the widget to derive different kinds of actions. There is a new initiative, Open UI Data,[13] that aims to use TESTAR with technical API to automatically generate screenshot images paired with JSON files containing information about widgets and their coordinates on the screenshot. The idea is then to use these UI data to train image recognition algorithms for better accuracy.

In Section 4.2, we described how actions were derived for detected web widgets. However, some web applications support customized classes that lead to non-native elements or native elements that by default would not be recognized as being interactive. Consequently, TESTAR is not able to derive actions automatically, if any. These have to be added manually. For that reason, the current

[12]https://opencv.org
[13]https://github.com/openuidata/openuidataset

DOI: 10.1002/stvr

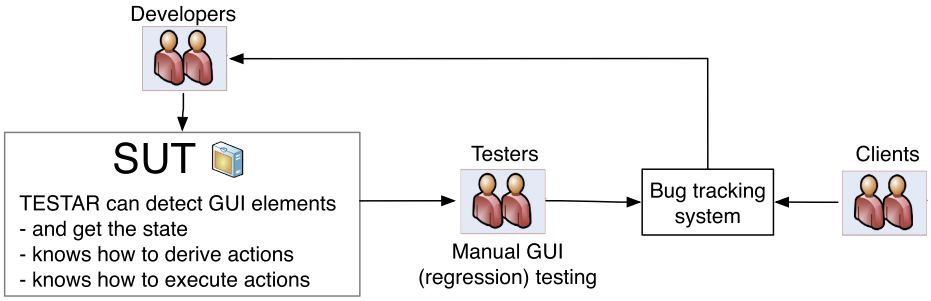

Figure 13. Similar components and interactions of the cases for generalization through architectural analogy. GUI, graphical user interface; SUT, system under test.

WebDriver implementation requires an initial manual overhead; we need to inspect the elements of the customized classes and add those with which we want TESTAR to interact. A possible future research topic would be to automatically detect or deduce which non-native elements can be interacted with and how. That would make the initial set-up of TESTAR a lot easier for web applications.

The development and testing of the WebDriver implementation was done on the Windows operating systems with Chrome. In the future, the support for more browsers and platforms is needed to enable research on cross-browser or cross-platform testing, automatically detecting changes in behaviour between browsers or platforms.

Changing the level of interaction with SUTs, we can study the possibility of using TESTAR to interact with extended or virtual reality 3D environments through designed frameworks, use autonomous agents intended to test environments by achieving goals or use operating system interfaces to interact at operating system level. This would require research on more advanced actions and more challenging state information.

8.2. Learning state models

State space explosion is a major challenge in learning GUI state models [4]. GUI state composes of all the screens visible for the user, all the widgets of each screen and all the properties and values of each widget. In addition, the previous states of the GUI might affect the behaviour and therefore might be relevant for deducing the current GUI state. We are currently researching how using different sets of GUI properties and including the previous states for calculating the abstract states will affect the inferred state model. Future directions for this research could include automatically selecting a suitable level of abstraction for a specific SUT. Nondeterminism is an issue when using the inferred state models for action selection. That means that selecting an action in a state does not always trigger the same result, and that is a problem if the action selection algorithm tries to navigate the state graph towards a specific state.

We are also implementing functionality to compare the models of subsequent SUT versions to automatically detect changes between SUT versions, already mentioned in Section 5.3. The first step would be manually analysing the reported changes and deciding if the change was intentional or a bug. For that purpose, a web-based GUI could provide an easier way for the user to go through the detected changes and visually inspect if the changes were intentional evolution of the application or unintentional regression bugs. The report should include a direct link to the web GUI of the reported model comparison. In future, machine learning could be used to learn what kind of changes are actually important and should be reported for manual analysis. The detected changes could be used for other purposes too, for example, reporting progress of the development or directing testing activities into changed areas of the SUT.

8.3. Recording models

For now, with TESTAR in the RECORD mode, a tester can record sequences that can be later replayed in the REPLAY mode. These manually recorded sequences are not stored in the model; they are just

DOI: 10.1002/stvr

for the tester to get familiar with the sequences of TESTAR. Functionality for recording manual or otherwise executed actions into the model would be a valuable extension. This way, we can let a user operate with the SUT in a way he or she would normally do, and we could record these SUT-specific actions by adding them to the state model. This could make it easier to teach TESTAR how to reach all parts of the SUT, instead of configuring the behaviour into SUT-specific TESTAR protocol Java class. We could also execute any existing script-based tests and reuse all the executed actions in test generation by recording them into the model.

Because this functionality would allow us to capture user stories into the state model, we could use this information to create an action selection algorithm that would associate some prioritization on these recorded actions to increase the possibility of these actions being executed. The idea is not to replay the exact same test sequence but to navigate semi-randomly through the user-stories functionality trees of the SUT using the state model.

Some SUT functionality flow that users have to execute with a combination of actions to reach some goal can involve a high number of possible elements to interact or configurations to achieve. This may be due to the fact that the user experience is not correctly defined or that the large number of available elements makes an application have a high GUI configuration complexity. If the state models would include the user interactions from manual or scripted testing, all the information about existing widgets and non-executed actions during testing would also be saved in the model. This information could allow us to obtain metrics for measuring the complexity of the GUI interactions, based on the number of widgets required for a specific functionality flow.

8.4. Actions

TESTAR's action selection can be improved by using artificial intelligence, machine learning or search-based techniques to learn what is the best action to execute next. As was already described in Section 6.4, related to the research already done on using reinforcement learning, more research is needed to invent new reward functions and policies for different goals of testing. In the context of the search-based techniques, different fitness functions need to be researched that, when optimized, lead to better testing. But also different search-based algorithms can be explored to see which behaves best for GUI testing. Or we can optimize for novelty without fitness.

Another interesting topic would be to investigate whether existing script-based tests or end user behaviour could guide us during the learning of the best action to execute next. The TESTAR model recording (listener) mode can help us capture the actions used in test scripts or learn from what real users would do.

Besides action selection, more research is needed to investigate how to generate better actions. For example, generating better inputs is an important future research topic. The goal is to automatically generate meaningful input values for text fields, for example, by deducting the type of the expected input from the neighbouring widgets, or generating potentially harmful input, for example, by using lists like the Big List of Naughty Strings.[14]

Finally, future work is to see if actions can be selected in a way that they establish the execution of application independent functionalities, or GUI patterns, that are defined in [74].

8.5. Test adequacy criteria

One of the general challenges in testing is the availability of appropriate *test adequacy indicators* to objectively justify the decision when we have tested enough [120]. These will enable us to assess the effectiveness of testing with TESTAR, compare different action selection mechanisms and define stopping criteria.

An effective test suite is the one that is able to detect the most errors. However, when testing, we never know how many and what errors there are in a system. (If we would, testing would no longer be necessary.) Moreover, knowing which of these errors would be the most important to find depends on the context in which the SUT is executed. Because there is no way to have the real measure of quality, *surrogate criteria* will have to suffice, that is, criteria of which we know, think or

[14]https://github.com/minimaxir/big-list-of-naughty-strings

DOI: 10.1002/stvr

hope that they correlate to the real criteria, like coverage or mutation scores. We have seen in Section 5.4 that there is some related work, but much more research is needed because useful indicators for GUI testing are unfortunately still scarce and lack empirical evaluation, especially for more recent and complex GUIs.

In future, we will research and evaluate different kinds of test metrics and stopping criteria. A possible stopping criterion can be defined using the saturation effect [121], a phenomenon that describes how the testing process may stop finding new faults in the SUT after a certain point. A solution, described by Tramontana *et al*. [122], requires the independent execution of several random testing sessions. The testing process is considered completed once all the testing sessions have run and the stopping criterion has been reached. In each testing session, a test sequence is generated and a testing target is measured. Two stopping criteria are provided based on comparison between the sets of testing targets covered by each of the test sequences generated in a testing session. If a stopping criterion is satisfied, it is assumed that a saturation point has been reached. For instance, Tramontana *et al*. use code coverage as a testing target in their GUI testing experiment. Several test sequences are executed, and depending on the stopping criterion used, if all the test sequences cover the same lines of code, or if one of the test sequences covers the union of the lines of code covered by the other test sequences, the testing process can be stopped. There is a challenge in applying this stopping criterion more generally in GUI testing, because measuring code coverage is not always easy for systems that consist of multiple backend systems that are, for example, used through a web GUI. Code coverage might not even be enough as a testing target, and experiments carried out by Tramontana *et al*. show a SUT that did not reach the saturation point, as specific code was not covered because the input necessary for its execution was not introduced.

Another possible stopping criteria could be defined with the model inference. When no more new states are found, and all the known actions have been executed at least once, it could mean that a saturation point in exploration has been reached. One option for future research could be, that, after that saturation point, another action selection algorithm is used, that is, changing from GUI exploration into combinatorial testing, trying to find new state transitions with a different order of executing actions in a specific state or a different order of executing state transitions.

8.6. Oracle problem

The challenge of automating oracles is known as the 'test oracle problem' [15-17]. Automation of oracles is one of the most important parts needed for success of test automation and tools like TESTAR. Usually, in scripted test automation, the oracle invocation is interleaved with the test case execution [123], and test oracle is defined based on the expected behaviour, resulting from the actions defined before the oracle invocation. In scriptless testing, test steps are generated during the test execution and therefore cannot be predicted in the same way. Each test oracle has to be defined as an agent that is triggered if certain conditions are fulfilled. In TESTAR, we distinguish online and offline oracles.

The work in [76] is the first step towards defining reusable offline oracles for one specific category of quality characteristics, that is, accessibility. Future work can look into oracles for categories of systems (i.e. different enterprise resource planning systems normally share many similarities), or categories of functionalities that occur in different systems (i.e. logon and shopping cart). Also, we could think of defining oracles as general properties within these categories, such that the inferred state model can be model checked to hold them or contradict them.

A new type of online oracles that can be researched are those detecting layout issues by comparing texts of widgets through image recognition and technical APIs. This could, for example, also be used for internationalization. The translations for using a GUI application in another language are often done without actually knowing the size of the widget where the text is going to be shown. Sometimes, the translations result in issues because the text does not fit, either breaking the layout or not showing all the intended text. We could define generic state oracles that detect these translation and layout issues by comparing texts of widgets through image recognition and technical APIs. The hypothesis would be that the technical API should have the whole text, even if it would not fit into the widget, and image recognition can be used to detect what is actually shown on the GUI.

DOI: 10.1002/stvr

Another future research aims at finding new ways for mining potential temporal test oracles from the inferred state model. These test oracles could be specified in linear temporal logic, and the inferred state model could be transformed into a format that allows model checking with an existing third-party tool. Then model checking could be used to check whether the potential test oracles are true in all of the existing sequences. The user should check the practical validity of the potential test oracles before actually using them for testing. For that reason, visualization is an important part for getting the user to understand what is proposed.

8.7. Visualization and test reporting

Currently, TESTAR generates a very simple HTML report for each executed test sequence as local files. The usability of the tool could be improved by generating more elegant and modern web-based reports. Parallel test execution should be also taken into account, so probably using a database instead of files would be a better option. There is a lot of information and details that should be available if needed, but the report should be layered in a way that starts with a dashboard, an overview of the whole test session, highlighting if failures were found and providing quick links to the found failures. Another possible direction of research, combining visualization and test reporting, would be visually showing what was covered with scriptless testing.

Analysing a long sequence that found a failure can be difficult. One possible future research would be to automatically search for a shortest path to reproduce a failure. With the inferred state model, finding a shorter sequence should not be that challenging, but it is not trivial to detect if it is the same failure or another one. Also, just knowing whether the failure can be reproduced is useful knowledge.

In the future, a better visualization of the state model could be used for many purposes, as described in Section 5. One interesting direction might be trying to automatically generate a GUI flow diagram in form that allows visual inspection. Currently, in many companies, such flow diagrams are manually created. The state model could also be used to improve test reporting in many ways, for example, visualizing the found failures in the state model.

9. CONCLUSIONS

This paper contains a comprehensive description of the open-source tool TESTAR, an academic prototype that has been used for scriptless GUI testing research the past 8 years. The last few years, more and more people external to the project become interested in TESTAR. Literature explaining the positioning of the tool within GUI testing, and a detailed explanation of how it works together with an agenda for future research direction, has been frequently requested. With this article, we hope to meet this request and convince people to use TESTAR for their GUI testing-related research and work together towards an international research agenda. The more people start using the tool, the higher the chances that we will have an increasingly stable and open-source infrastructure in the near future.

ACKNOWLEDGEMENTS

In addition to the authors, many people have contributed to TESTAR throughout the years. We will list them in alphabetical order: Sami Ahonen, Francisco Almenar Pedrós, Alessandra Bagnato, Salmi Baharom, Sebastian Bauersfeld, Etienne Brosse, Govert Buijs, Ernesto Calas Blasco, Hatim Chahim, Nelly Condori, Wouter Cox, Martin Deiman, Mehmet Duran, Anna Esparcia-Alcazar, Joan Fons i Cors, Floren de Gier, Stijn de Gouw, Marion de Groot, Conny Hageluken, Mark Harman, Davy Kager, Eddy Kivits, Peter Kruse, Melvin van der Kuijl, Yvan Labiche, Jean-Marc Maas, Hector Martinez Martinez, Tycho Menting, Perttu Moilanen, Mirella Oreto Martínez Murillo, Carlos Ortega, Mauro Pezze, Dietmar Pfahl, Tomi Piirainen, Wishnu Prasetya, Enrique Reverón, Antonio de Rojas, Urko Rueda, Carlo Sengers, Guy Thieuws, Ismael Torres Boigues, Ramón de Vries, Joachim Wegener, Adri Weijling. This work has been funded by the following projects: FITTEST (FP7 Information and Communication Technologies, 257574), SHIP (EACEA/A2/

DOI: 10.1002/stvr

UHB/CL 554187), PERTEST (TIN2013-46928-C3-1-R), TESTOMAT (ITEA3, 16032), DECODER (H2020, 824231), iv4XR (H2020, 856716), TAILOR (H2020, 952215) and IVVES (ITEA3, 18022).

DATA AVAILABILITY STATEMENT

Data sharing was not applicable to this article as no datasets were generated or analysed during the current study.

REFERENCES

1. https://www.tricentis.com/resources/software%2Dfail%2Dwatch%2D5th%2Dedition/ (accessed jan 2021).
2. Campos J, Arcuri A, Fraser G, Abreu R. Continuous test generation: enhancing continuous integration with automated test generation. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. ACM: New York, NY, USA, 2014; 55–66.
3. Fraser G, Arcuri A. A large scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 2014; **24**(2): 8.
4. Aho P, Vos TEJ. Challenges in automated testing through graphical user interface. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE Computer Society: Los Alamitos, CA, USA, 2018; 118–121. https://doi.ieeecomputersociety.org/10.1109/ICSTW.2018.00038
5. Pezzè M, Rondena P, Zuddas D. Automatic GUI testing of desktop applications: an empirical assessment of the state of the art. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM: New York, NY, USA, 2018; 54–62. http://doi.acm.org/10.1145/3236454.3236489
6. Cheng L, Chang J, Yang Z, Wang C. GUICat: GUI testing as a service. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*: Singapore, 2016; 858–863.
7. Bauersfeld S, Vos TEJ. User interface level testing with TESTAR; what about more sophisticated action specification and selection? In *Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE 2014)*. L'Aquila, Italy, 2014; 60–78.
8. Bauersfeld S. GUIdiff – a regression testing tool for graphical user interfaces. In *2013 IEEE Sixth international Conference on Software Testing, Verification and Validation*. Luxembourg, 2013; 499–500.
9. Coppola R, Ardito L, Torchiano M. Fragility of layout-based and visual GUI test scripts: an assessment study on a hybrid mobile application. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating Test Case Design, Selection, and Evaluation*. ACM: New York, NY, USA, 2019; 28–34. http://doi.acm.org/10.1145/3340433.3342824
10. Coppola R, Ardito L, Torchiano M, Morisio M. Mobile testing: new challenges and perceived difficulties from developers of the Italian industry, 2019. IT PROFESSIONAL.
11. Alégroth E, Feldt R, Kolström P. Maintenance of automated test suites in industry: an empirical study on visual GUI testing. *Information and Software Technology*. 2016; **73**: 66–80. http://www.sciencedirect.com/science/article/pii/S0950584916300118
12. Berner S, Weber R, Keller RK. Observations and lessons learned from automated testing. In *Proceedings 27th International Conference on Software Engineering, 2005. ICSE 2005*: St. Louis, MO, USA, 2005; 571–579.
13. Leotta M, Clerissi D, Ricca F, Tonella P. Capture-replay vs. programmable web testing: an empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*: Koblenz, Germany, 2013; 272–281.
14. Vos TEJ, Aho P. Searching for the best test. In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*. Buenos Aires, Argentina, 2017; 3–4.
15. Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S. The oracle problem in software testing: a survey. *IEEE Transactions on Software Engineering*. 2015; **41**(5): 507–525.
16. Jahangirova G. Oracle problem in software testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM: New York, NY, USA, 2017; 444–447. http://doi.acm.org/10.1145/3092703.3098235
17. Oliveira RAP, Kanewala U, Nardi PA. Automated test oracles: state of the art, taxonomies, and trends. *Advances in Computers*. 2014; **95**: 113–199. https://www.sciencedirect.com/science/article/pii/B9780128001608000036
18. Vos TEJ, Tonella P, Wegener J, Harman M, Prasetya W, Puoskari E, Nir–Buchbinder Y. Future internet testing with fittest. In *15th European Conference on Software Maintenance and Reengineering*. Oldenburg, Germany, 2011; 355–358.
19. Vos TEJ, Esparcia AI. Software testing innovation alliance – the SHIP project. In *Joint proceedings of the doctoral symposium and projects showcase held as part of STAF 2016 co-located with Software Technologies: Applications and Foundations (STAF 2016)*: Vienna, Austria, 2016; 65–71. http://ceur-ws.org/Vol-1675/paper8.pdf
20. Vos TEJ, Kruse PM, Condori-Fernández N, Bauersfeld S, Wegener J. TESTAR: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design*. 2015; **6**(3): 46–83. https://doi.org/10.4018/IJISMD.2015070103

21. Alégroth E, Feldt R. 2014. Industrial application of visual GUI testing: lessons learned. In *Continuous Software Engineering*, Bosch J (ed.). Springer International Publishing: Cham; 127–140. https://doi.org/10.1007/978-3-319-11283-1_11
22. Aho P, Alégroth E, Oliveira RAP, Vos TEJ. Evolution of automated regression testing of software systems through the graphical user interface. In *The First International Conference on Advances in Computation, Communications and Services (ACCSE 2016)*. Valencia, Spain, 2016; 16–21.
23. Rapise. [Online; accessed 20-12-2019]. https://www.inflectra.com/Rapise/
24. Squish. [Online; accessed 20-12-2019]. https://www.froglogic.com/squish/
25. Ranorex. [Online; accessed 20-12-2019]. https://www.ranorex.com/
26. Autoit. https://www.autoitscript.com/site/autoit/, [Online; accessed 20-12-2019].
27. Selenium. [Online; accessed 20-12-2019]. https://selenium.dev/
28. Sikulix. [Online; accessed 20-12-2019]. http://sikulix.com/
29. Eyeautomate. https://eyeautomate.com/, [Online; accessed 20-12-2019].
30. Aho P, Kanstrén T, Räty T, Röning J. Automated extraction of GUI models for testing. *Advances in Computers*. 2015; **95**: 49–112. https://doi.org/10.1016/B978-0-12-800160-8.00002-4
31. Choudhary SR, Zhao D, Versee H, Orso A. Water: web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*. Association for Computing Machinery: New York, NY, USA, 2011; 24–29. https://doi.org/10.1145/2002931.2002935
32. Stocco A, Yandrapally R, Mesbah A. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM: New York, NY, USA, 2018; 503–514. http://doi.acm.org/10.1145/3236024.3236063
33. Gao Z, Chen Z, Zou Y, Memon AM. SITAR: GUI test script repair. *IEEE Transactions on Software Engineering*. 2016; **42**(2): 170–186.
34. Leotta M, Stocco A, Ricca F, Tonella P. Robula+: an algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*. 2016; **28**(3): 177–204. https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1771
35. Silva JL, Campos J, Paiva A. Model-based user interface testing with spec explorer and concurtasktrees. *Electronic Notes in Theoretical Computer Science*. 2008; **208**: 77–93.
36. Chinnapongse V, Lee I, Sokolsky O, Wang S, Jones P. 2009. Model-based testing of GUI-driven applications. In *Software Technologies for Embedded and Ubiquitous Systems*, Lee S, Narasimhan P (eds). Springer Berlin Heidelberg: Berlin, Heidelberg; 203–214.
37. Moreira RMLM, Paiva A, Nabuco M, Memon AM. Pattern-based GUI testing: bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability*. 2017; **27**(3): e1629. https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1629
38. Andrews AA, Offutt J, Alexander RT. Testing web applications by modeling with FSMs. *Software & Systems Modeling*. 2005; **4**: 326–345. https://doi.org/10.1007/s10270-004-0077-7
39. Takala T, Katara M, Harty J. Experiences of system-level model-based GUI testing of an android application. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. Berlin, Germany: IEEE Computer Society, 2011; 377–86. https://doi.org/10.1109/ICST.2011.11
40. Miao Y, Yang X. An FSM based GUI test automation model. In *2010 11th International Conference on Control Automation Robotics Vision*: Singapore, 2010; 120–126.
41. Memon AM, Banerjee I, Nagarajan A. GUI ripping: reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings*. Victoria, Canada, 2003; 260–269.
42. Bertolino A, Polini A, Inverardi P, Muccini H. Towards anti-model-based testing. In *Fast Abstract in The Int'l. Conf. on Dependable Systems and Networks, DSN*. Florence, Italy, 2004; 124–125.
43. Aho P, Menz N, Räty T, Schieferdecker I. Automated Java GUI modeling for model-based testing purposes. In *2011 Eighth International Conference on Information Technology: New Generations*: Las Vegas, NV, USA, 2011; 268–273.
44. Mesbah A, van Deursen A, Lenselink S. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transaction Web*. 2012; **6**(1): 1–30. https://doi.org/10.1145/2109205.2109208
45. Amalfitano D, Fasolino AR, Tramontana P, Amatucci N. Considering context events in event-based testing of mobile applications. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*: Luxembourg, Luxembourg, 2013; 126–133.
46. Marchetto A, Tonella P, Ricca F. State-based testing of Ajax web applications. In *2008 1st International Conference on Software Testing, Verification, and Validation*: Lillehammer, Norway, 2008; 121–130.
47. Hofer B, Peischl B, Wotawa F. GUI savvy end-to-end testing with smart monkeys. In *ICSE Workshop on Automation of Software Test, 2009. AST '09*: Vancouver, BC, Canada, 2009; 130–137.
48. Nyman N. Using monkey test tools – how to find bugs cost-effectively through random testing. *Software Testing & Quality Engineering*. 2000: 18–21.
49. Hertzfeld A. *Revolution in the Valley (hardcover)*. Sebastopol, CA, USA: O' Reilly & Associates, Inc., 2004.
50. Lipner S. The trustworthy computing security development lifecycle. In *Proceedings of the 20th Annual Computer Security Applications Conference*. IEEE Computer Society: Washington, DC, USA, 2004; 2–13. https://doi.org/10.1109/CSAC.2004.41
51. The (Google) monkey. [Online; accessed 25-12-2019].

52. Girard E, Rault JC. A programming technique for software reliability. In *IEEE Symposium on Computer Software Reliability*. New York, USA, 1973. 44–50.
53. Thayer TA, Lipow M, Nelson EC. *Software Reliability*. North-Holland Pub. Co: Amsterdam, 1978.
54. Myers GJ. *The Art of Software Testing*. Hoboken, NJ, USA: John Wiley and Sons, 1979.
55. Beizer B. *Software Testing Techniques* (2nd edn.) Van Nostrand Reinhold Co.: New York, NY, USA, 1990.
56. Duran JW, Ntafos SC. An evaluation of random testing. *IEEE TSE*. 1984; **SE-10**(4): 438–444.
57. Hamlet D, Taylor R. Partition testing does not inspire confidence. In *Banff, AB, Canada*, 1988; 206–215.
58. Weyuker EJ, Jeng B. Analyzing partition testing strategies. *IEEE TSE*. 1991; **17**(7): 703–711.
59. Arcuri A, Briand L. Adaptive random testing: an illusion of effectiveness? In *International Symposium on Software Testing and Analysis*: NY, USA, 2011; 265–275. http://doi.acm.org/10.1145/2001420.2001452
60. Chen TY, Yu YT. On the relationship between partition and random testing. *IEEE Transactions on Software Engineering*. 1994; **20**(12): 977–980. https://doi.org/10.1109/32.368132
61. Gutjahr WJ. Partition testing vs. random testing: the influence of uncertainty. *IEEE Transactions on Software Engineering*. 1999; **25**(5): 661–674. https://doi.org/10.1109/32.815325
62. Tsoukalas MZ, Duran JW, Ntafos SC. On some reliability estimation problems in random and partition testing. *IEEE Transactions on Software Engineering*. 1993; **19**(7): 687–697. https://doi.org/10.1109/32.238569
63. Arcuri A, Iqbal MZ, Briand L. Random testing: theoretical results and practical implications. *IEEE TSE*. 2012; **38**(2): 258–277.
64. Böhme M, Paul S. A probabilistic analysis of the efficiency of automated software testing. *IEEE TSE*. 2016; **42**(4): 345–360.
65. Amalfitano D, Amatucci N, Memon AM, Tramontana P, Fasolino AR. A general framework for comparing automatic testing techniques of android mobile apps. *Journal of Systems and Software*. 2017; **125**: 322–343. http://www.sciencedirect.com/science/article/pii/S016412121630259X
66. Bertolini C, Peres G, d' Amorim M, Mota A. An empirical evaluation of automated black box testing techniques for crashing GUIs. In *2009 International Conference on Software Testing Verification and Validation*: Denver, CO, USA, 2009; 21–30.
67. Aho P, Suarez M, Kanstrén T, Memon AM. Murphy tools: utilizing extracted GUI models for industrial software testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*: Cleveland, OH, USA, 2014; 343–348.
68. Appium. [Online; accessed 25-12-2019].
69. Martínez M, Esparcia-Alcázar AI, Vos TEJ, Aho P, i Cors JF. 2018. Towards automated testing of the internet of things: results obtained with the TESTAR tool. In *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems*, Margaria T, Steffen B (eds). Springer International Publishing: Cham; 375–385.
70. UIA Windows. [Online; accessed 25-12-2019]. https://docs.microsoft.com/en-us/windows/win32/winauto/uiauto-entry-propids
71. Mesbah A, Van Deursen A. Invariant-based automatic testing of Aajax user interfaces. In *2009 IEEE 31st International Conference on Software Engineering*, IEEE: Vancouver, BC, Canada, 2009; 210–220.
72. Dallmeier V, Burger M, Orth T, Zeller A. Webmate: a tool for testing web 2.0 applications. In *Proceedings of the Workshop on Javascript Tools*. ACM: New York, NY, USA, 2012; 11–15. http://doi.acm.org/10.1145/2307720.2307722
73. Nguyen B, Robbins B, Banerjee I, Memon AM. Guitar: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*. 2013; **21**(1): 65–105. https://doi.org/10.1007/s10515-013-0128-9
74. Mariani L, Pezzè M, Zuddas D. Augusto: exploiting popular functionalities for the generation of semantic GUI tests with oracles. In *Proceedings of the 40th International Conference on Software Engineering*. Association for Computing Machinery: New York, NY, USA, 2018; 280–290. https://doi.org/10.1145/3180155.3180162
75. Leonardo M, Mauro P, Oliviero R, Mauro S. AutoBlackTest: A Tool for Automatic Black-box Testing. *Proceedings of the 33rd International Conference on Software Engineerin*. ICSE '11, New York, NY, USA: ACM; 2011. 1013–1015. https://doi.acm.org/10.1145/1985793.1985979
76. de Gier F, Kager D, de Gouw S, Vos TEJ. Offline oracles for accessibility evaluation with the TESTAR tool. In *2019 13th International Conference on Research Challenges in Information Science (RCIS)*. Brussels, Belgium, 2019; 1–12.
77. https://gephi.org/, 2019. Accessed: 2019-05-04.
78. https://www.graphviz.org/, 2019. Accessed: 2019-05-06.
79. https://cytoscape.org/, 2019. Accessed: 2019-05-07.
80. Memon AM, Soffa ML, Pollack M. Coverage criteria for GUI testing. *SIGSOFT Softw. Eng. Notes*. 2001; **26**(5): 256–267. http://doi.acm.org/10.1145/503271.503244
81. Xie Q, Memon AM. Studying the characteristics of a "good" GUI test suite. In *2006 17th IEEE International Symposium on Software Reliability Engineering*. IEEE Computer Society: Los Alamitos, CA, USA, 2006; 159–168. https://doi.ieeecomputersociety.org/10.1109/ISSRE.2006.45
82. Strecker J, Memon AM. Relationships between test suites, faults, and fault detection in GUI testing. In 2008 1st International Conference on Software Testing, Verification, and Validation. Lillehammer, Norway, 2008; 12–21.
83. Carino S, Andrews JH. Evaluating the effect of test case length on GUI test suite performance. In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*: Florence, Italy, 2015; 13–17.
84. McMaster S, Memon AM. Call-stack coverage for GUI test suite reduction. *IEEE Transactions on Software Engineering*. 2008; **34**(1): 99–115. https://doi.org/10.1109/TSE.2007.70756

85. Bryce RC, Memon AM. Test suite prioritization by interaction coverage. In *Workshop on Domain Specific Approaches to Software Test Automation: In Conjunction with the 6th ESEC/FSE Joint Meeting*. Association for Computing Machinery: New York, NY, USA, 2007; 1–7. https://doi.org/10.1145/1294921.1294922
86. Zhao L, Cai K. Event handler-based coverage for GUI testing. In *2010 10th International Conference on Quality Software*. Zhangjiajie, China, 2010; 326–331.
87. Yuan X, Cohen M, Memon AM. GUI interaction testing: incorporating event context. *IEEE Transactions on Software Engineering*. 2011; **37**(4): 559–574.
88. Oliveira RAP, Alégroth E, Gao Z, Memon A. Definition and evaluation of mutation operators for GUI-level mutation analysis. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Graz, Austria, 2015; 1–10.
89. Beierle N, Kruse PM, Vos TEJ. GUI-profiling for performance and coverage analysis. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2017; 28–31.
90. Esparcia AI, Almenar F, Martínez M, Rueda U, Vos TEJ. Q-learning strategies for action selection in the TESTAR automated testing too. In *6th International Conference on Metaheuristics and Nature Inspired Computing (META 2016)*. Marrakech, Morocco, 2016; 130–137.
91. Esparcia AI, Almenar F, Vos TEJ, Rueda U. Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the TESTAR tool. *Memetic Computing*. 2018; **10**(3): 257–265. https://doi.org/10.1007/s12293-018-0263-8
92. Esparcia-Alcázar AI, Almenar F, Rueda U, Vos TEJ. Evolving rules for action selection in automated testing via genetic programming – a first approach. In *Applications of Evolutionary Computation*, Squillero G, Sim K (eds). Cham, Switzerland: Springer International Publishing, 2017; 82–95.
93. Rodriguez MA. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*: Pittsburgh, PA, USA, 2015; 1–10. https://doi.org/10.1145/2815072.2815073
94. Korn P, Martínez Normand L, Pluke M, Snow-Weaver A, Vanderheiden G. Guidance on applying WCAG 2.0 to non-web information and communications technologies (WCAG2ICT), 2013. http://www.w3.org/WAI/GL/2013/WD-wcag2ict-20130905/, Editors' Draft, work in progress.
95. Bauersfeld S, Vos TEJ. A reinforcement learning approach to automated GUI robustness testing. In *In Fast Abstracts of the 4th Symposium on Search-Based Software Engineering (SSBSE 2012)*. IEEE: Riva del Garda, Italy, 2012; 7–12.
96. David A, Md Khorrom K, Koppula K, Ren{¥'e}e B. Reinforcement Learning for Android GUI Testing. *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. New York, NY, USA: ACM; 2018. 2–8. https://doi.org/10.1145/3278186.3278187
97. Mariani L, Pezze M, Riganelli O, Santoro M. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012. 81–90. https://doi.org/10.1109/ICST.2012.8
98. Christian D, Borges J, Nataniel P, Andreas Z. Learning User Interface Element Interactions. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019, New York, NY, USA: ACM; 2019. 296–306. http://doi.acm.org/10.1145/3293882.3330569
99. Koroglu Y, Sen A, Muslu O, Mete Y, Ulker C, Tanriverdi T, Donmez Y. QBE: QLearning-Based Exploration of Android Applications. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society; 2018. 105–115. https://doi.org.10.1109/ICST.2018.00020
100. Gu T, Cao C, Liu T, Sun C, Deng J, Ma M, Lü J. AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017. 103–114. https://doi.org/10.1109/ICSME.2017.72
101. van Otterlo M, Wiering M. 2012. Reinforcement learning and Markov decision processes. In *Reinforcement Learning: State-of-the-art*,Wiering M, van Otterlo M (eds). Springer Berlin Heidelberg: Berlin, Heidelberg; 3–42. https://doi.org/10.1007/978-3-642-27645-3_1
102. Bauersfeld S, Wappler S, Wegener J. A metaheuristic approach to test sequence generation for applications with a GUI. In *Proceedings of the Third International Conference on Search Based Software Engineering*. Springer-Verlag: Berlin, Heidelberg, 2011; 173–187. http://dl.acm.org/citation.cfm%3Fid%3D2042243.2042267
103. Luke S. *Essentials of Metaheuristics* (2nd edn.) Lulu, 2013. Available for free at http://cs.gmu.edu/%5C%7Esean/book/metaheuristics/
104. Carino S, Andrews JH. Dynamically testing GUIs using ant colony optimization (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015; 138–148.
105. Lehman J, Stanley KO. Abandoning objectives: evolution through the search for novelty alone. *Evolutionary Computation*. 2011; **19**(2): 189–223. https://doi.org/10.1162/EVCO_a_00025
106. Vos TEJ, Marín B, Escalona MJ, Marchetto A. A methodological framework for evaluating software testing techniques and tools. In *12th International Conference on Quality Software*: Xi'an, China, 2012; 230–239.
107. Vos TEJ. Evolutionary testing for complex systems. *ERCIM News*. 2009; **2009**(78):45–46.
108. Vos TEJ. Continuous evolutionary automated testing for the future internet. *ERCIM News*. 2010; **2010**(82): 50–51.
109. Host M, Runeson P. Checklists for software engineering case study research. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society: Washington, DC, USA, 2007; 479–481. https://doi.org/10.1109/ESEM.2007.29

110. Kitchenham B, Linkman S, Law D. Desmet: a methodology for evaluating software engineering methods and tools. *Computing Control Engineering Journal*. 1997; **8**(3): 120–126.
111. Kitchenham B, Pickard LM, Pfleeger SL. Case studies for method and tool evaluation. *IEEE Software*. 1995; **12**(4): 52–62.
112. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*. 2009; **14**(2): 131–164.
113. Bauersfeld S, Vos TEJ, Condori-Fernández N, Bagnato A, Brosse E. Evaluating the TESTAR tool in an industrial case study. In *Torino, Italy*, 2014; 4. http://doi.acm.org/10.1145/2652524.2652588
114. Ricós FP, Aho P, Vos T, Boigues IT, Blasco EC, Martínez HM. Deploying TESTAR to enable remote testing in an industrial CI pipeline: a case-based evaluation, 2020; 543–557.
115. Bauersfeld S, de Rojas A, Vos TEJ. Evaluating rogue user testing in industry: an experience report. In *2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS)*. Marrakech, Morocco, 2014; 1–10.
116. Chahim H, Duran M, Vos TEJ, Aho P, Condori Fernandez N. 2020. Scriptless testing at the GUI level in an industrial setting. In *Research Challenges in Information Science*, Dalpiaz F, Zdravkovic J, Loucopoulos P (eds). Springer International Publishing: Cham; 267–284.
117. Aho P, Vos TEJ, Ahonen S, Piirainen T, Moilanen P, Pastor Ricos F. 2019. Continuous piloting of an open source test automation tool in an industrial environment. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*. Cáceres, Spain: Sistedes; 1–4.
118. Martinez M, Esparcia AI, Rueda U, Vos TEJ, Ortega C. 2016. Automated localisation testing in industry with TESTAR. In *Testing Software and Systems*, Wotawa F, Nica M, Kushik N (eds). Springer International Publishing: Cham; 241–248.
119. Wieringa R, Daneva M. Six strategies for generalizing software engineering theories. *Science of Computer Programming*. 2015; **101**: 136–152.
120. Belli F. Finite state testing and analysis of graphical user interfaces. In *Proceedings 12th International Symposium on Software Reliability Engineering*: Hong Kong, China, 2001; 34–43.
121. Lyu MR, et al. *Handbook of Software Reliability Engineering*, vol. 222. IEEE Computer Society Press: CA, 1996.
122. Tramontana P, Amalfitano D, Amatucci N, Memon A, Fasolino AR. Developing and evaluating objective termination criteria for random testing. *ACM Transactions on Software Engineering and Methodology*. 2019; **28**(3): 1–52. https://doi.org/10.1145/3339836
123. Memon AM. GUI testing: pitfalls and process. *Computer*. 2002; **35**(8): 87–88.