State model inference through the GUI using run-time test generation

Ad Mulders[1], Olivia Rodriguez Valdes[1], Fernando Pastor Ricós[2], Pekka Aho[1], Beatriz Marín[2], and Tanja E.J. Vos[1,2]

[1] Open Universiteit, The Netherlands
[2] Universitat Politècnica de València, Spain

Abstract. Software testing is an important part of engineering trustworthy information systems. End-to-end testing through Graphical User Interface (GUI) can be done manually, but it is a very time consuming and costly process. There are tools to capture or manually define scripts for automating regression testing through a GUI, but the main challenge is the high maintenance cost of the scripts when the GUI changes. In addition, GUIs tend to have a large state space, so creating scripts to cover all the possible paths and defining test oracles to check all the elements of all the states would be an enormous effort. This paper presents an approach to automatically explore a GUI while inferring state models that are used for action selection in run-time GUI test generation, implemented as an extension to the open source TESTAR tool. As an initial validation, we experiment on the impact of using various state abstraction mechanisms on the model inference and the performance of the implemented action selection algorithm based on the inferred model. Later, we analyse the challenges and provide future research directions on model inference and scriptless GUI testing.

Keywords: Model inference · Automated GUI testing · TESTAR tool

1 Introduction

The world around us is strongly connected through software and information systems. Graphical User Interface (GUI) represents the main connection point between software components and the end users, and can be found in most modern applications. Testing through GUI is an important way to prevent end users from experiencing the effects of software bugs. Although GUI testing can be done manually, it is a very time consuming and costly process [14]. There are various tools to capture or manually define scripts for regression testing of GUIs, but the main challenge is the high maintenance cost of the scripts when the GUI changes [25]. In addition, GUIs tend to have a large state space, so creating scripts to cover all the possible paths and defining test oracles to check all the elements of all the states would be an enormous effort.

Scriptless GUI testing aims to lower the maintenance costs compared to scripted testing. In scriptless testing, there are no scripts that define the sequences of test steps prior to test execution. Instead, the testing tool decides

on-the-fly during test execution which test steps are being executed, based on the action selection mechanism (ASM) being used. Although there are no test scripts to maintain, there might be some system specific instructions for the tool that require maintenance.

TESTAR (testar.org), an open-source tool for scriptless GUI testing, is based on agents that implement various ASMs. 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 select and execute an action to bring the SUT into another state. Various case studies have shown that TESTAR effectively complements the existing manual practices and can find undiscovered failures in a SUT with reasonably low setup costs [27]. However, TESTAR suffers from not knowing exactly what has been tested, failing to give test managers the information they need for decision making.

In [13, 27], the first steps are described towards an extension of TESTAR to infer state models during GUI exploration. This feature allows creating a map of where in the SUT the tool has been and what it has done during testing. De Gier et al. [13] use the model to define offline oracles (i.e., after testing) that consisted of querying the model for accessibility information. However, inferring state models can be purposeful for TESTAR in various other ways. For example, the inferred models can be used for TESTAR's ASMs during and after the model inference, or defining various types of offline oracles that are based on comparing models between different releases or versions of a System Under Test (SUT). The models could also serve as visual reference models for testers or users, or be valuable for model-based GUI testing (MBGT) tools [3]. MBGT has not been widely adopted, because creating the models requires modelling expertise and a lot of effort [7]. If we can infer even initial models, these problems might be (partially) solved.

There are a few existing approaches for inferring models during automated exploration of the GUI that are described in Section 2. However, automated GUI exploration is challenging, and existing tools are mostly academic prototypes or abandoned open source projects. State space explosion is still an open challenge for the inference of state-based models through GUI. Most programs with a GUI have a huge number of possible states, and to make the size of the models manageable, some information has to be abstracted away. It is challenging to define a suitable level of abstraction and find an equilibrium between the necessary expressiveness of the extracted models and the computational complexity [19]. Abstracting away too much information might make a model unsuitable for its purpose (i.e., ASM, MBGT, oracles, etc) and lose opportunities to discover faults and changes between versions. Abstracting away too little information might result in state space explosion, making the model less suitable for its purpose. Most of the related work does not explain in sufficient detail how they deal with the abstraction, raising questions whether their solutions are generally applicable or simply tailored for the applications used in validation.

The main contributions of this paper are:

– A description of the model inference functionality implemented into TESTAR.

– A new algorithm for ASM based on the inferred model.
– An initial validation of the test effectiveness of the new ASM in terms of code coverage and reached states.
– An initial validation of the approach by experimenting on how various abstraction mechanisms affect the inferred models.

The rest of the paper is organised as follows. Section 2 presents related work. Section 3 presents TESTAR and our approach to infer state models. Section 4 presents the use of models for action selection and experiments on the test effectiveness of the implemented ASM. Section 5 describes the experimentation to find out how various abstraction mechanisms affect the inferred models. Section 6 analyses the findings and challenges, and provides future research directions. Finally, Section 7 presents the main conclusions.

2 Related work

The earliest automated GUI testing tools were so called capture and replay (C&R) tools. Using them, a test engineer manually inputs all the interactions by using mouse and keyboard while having the tool recording the test case. Afterwards, the test sequence could be executed automatically as part of a regression test set. The main advantage of C&R tools is that they are easy to use and testers do not have to write test scripts by hand. The disadvantage is that the recorded scripts are fragile for GUI changes and maintenance of the scripts is costly since the out-dated test cases have to be manually recorded again [23].

Model-based GUI testing (MBGT) [3, 11, 16] aims to reduce the effort for creating and maintaining the test scripts by generating the test cases from a model. MBGT approaches require that the GUI and its expected behavior is modelled on a higher level of abstraction than the GUI itself. The modelling language should be understandable by a tool that uses it to automatically generate tests. An advantage of this type of testing is that it is possible to precisely specify the exact test specifications that a GUI should conform to. Another advantage is that when the GUI changes, the test scripts do not have to be manually updated. Instead, the model is updated and the scripts/tests are generated again. The main disadvantages are that model-based GUI testing approaches require a deep knowledge of the application domain and expert knowledge of formal modelling methods and languages to manually create a model of the GUI. Modelling requires also quite a lot of time and effort.

There are a few GUI testing approaches that allow automated GUI model inference, a.k.a., GUI ripping [20] or GUI reverse engineering [15]. We can roughly distinguish 3 forms of automated model discovery: (1) static analysis, (2) dynamic analysis and (3) a hybrid combination of both static and dynamic techniques [17, 4]. Model inference through static analysis uses the program's source code to infer a model of the GUI [26, 12]. Static techniques concentrate only on the structure of the GUI, not taking the run-time behaviour of the GUI into consideration in the model.

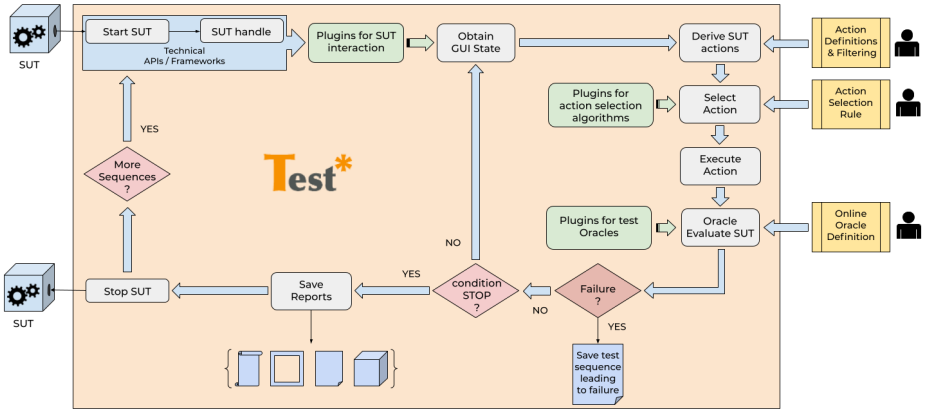


Fig. 1: TESTAR operational flow to generate sequences of (state,action)-pairs by starting up the SUT and continuously select and execute an action to bring the SUT into another state.

The dynamic model inference approaches analyse the GUI while the system is running [6]. To automatically explore the GUI, APIs or libraries are used to get access to all the GUI elements in a specific state of the application. To create a model, these tools are able to recognise whether the application is in a state that the tool has already visited before, or whether the state is being visited for the first time. Examples of tools using model inference through dynamic analysis are GUITAR [24], GUI Driver [5], Crawljax [21], Extended Ripper [8], GuiTam [22] and Murphy Tools [6]. Some approaches, e.g., [18], combine dynamic and static analysis for model extraction. As mentioned, the challenge that has to be solved for all these tools is to define a suitable level of abstraction for the model inference that ensures that the model is useful for its purpose (e.g., ASMs, offline oracles, visualisation of testing, etc). Most of the related work does not explain in sufficient detail how they deal with abstraction. In this paper, we will make a first attempt to research how the abstraction level affects the results when using the models.

3 State model inference for TESTAR

The operational flow of TESTAR is shown in Fig. 1. When the SUT has *started*, TESTAR captures the current *state of the GUI* using APIs like Windows Automation API (WUIA) (for desktop), Selenium Web Driver (for web), or the Java access bridge (for Swing). This (concrete) state consists of *all* the properties (that are available through the API) of all the widgets that are part of the GUI. Subsequently, to *derive* the actions that it is able to perform in that state, it cycles through all the widgets and adds all possible actions associated with the widgets to a pool. Sometimes, if the SUT includes custom widgets and the API does not detect all the widget attributes, the user has to provide TESTAR with some extra configuration to detect all the available actions correctly. From this

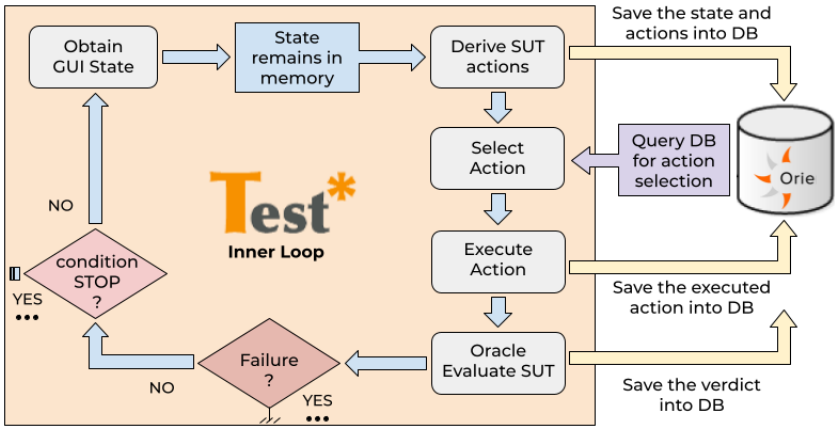


Fig. 2: TESTAR operational flow including model inference

action pool, one is *selected* by the ASM of TESTAR and *executed*. TESTAR has several ASMs available, for example based on random, prioritising new actions, or execution count [10].

After the action has been executed and the GUI has reached a new state, TESTAR will again capture the new state and derive, select and execute an action. This process repeats until the specified stop condition, for example the test sequence length or the occurrence of an error condition, is reached.

State abstraction is an important facet of scriptless GUI testing. TESTAR has an implementation to calculate state identifiers based on hashes over a selected set of widget attributes. This selected set defines the abstraction level. The abstraction level determines the number of different states TESTAR will distinguish. This can evidently influence test effectiveness and is related to the equilibrium explained in Section 1. To gather evidence about the suitable set of widget attributes for state abstraction, we run experiments that are described in Section 5.

TESTAR uses dynamic analysis techniques to infer a model. The flow for capturing the state model is depicted in Fig. 2. The state of the SUT is constantly saved into the OrientDB graph database together with available actions and the executed action. The state model can be queried by an ASM, like in Section 4, but also by a human, an offline oracle, or other MBGT approaches.

As indicated, the model will be built incrementally with subsequent TESTAR runs. All states (concrete and abstract) that are visited during a run are stored in the database. For analysis and reporting, the structure of the inferred model is divided into three layers as shown in Fig. 3.

The **top layer** is an abstract state model. It allows for example ASMs to use the model for action selection, or end users to analyse the behaviour of the SUT. Creating the abstract model requires the identification of unique states at a *suitable* abstraction level. As indicated, this means trying to avoid state space explosion, while simultaneously not losing the purposefulness of the model. Too abstract states can introduce non-determinism in the inferred model.

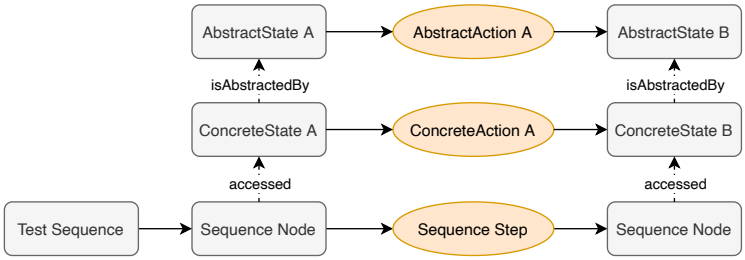


Fig. 3: Layered design of the state model

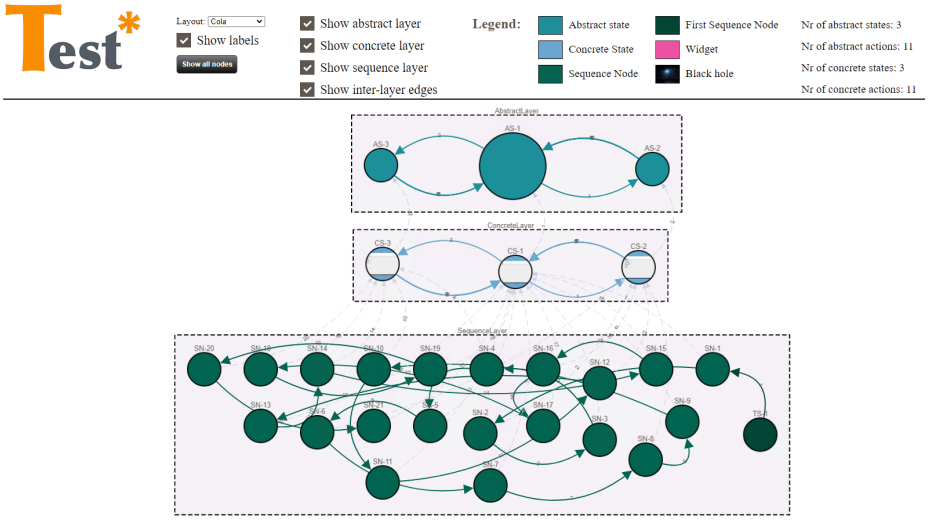


Fig. 4: Visualization of an example model inferred by TESTAR

The **mid layer** is the concrete model. This model contains all the information that can be extracted through the APIs used by TESTAR. The concrete state model will contain too many states to drive the execution of TESTAR or serve as a visual model for humans. It will serve as information storage, e.g., when a specific part of the abstract model requires deeper analysis. Each concrete state of this layer will be linked to an abstract one in the top layer, and each action will be linked to an abstract transition.

The **bottom layer** is the management layer, and its purpose will be to record meta information about the executed test sequences. Where the abstract and concrete layers describe the SUT, the management layer describes the execution of the tests in TESTAR. The individual test sequences will be linked to the concrete states and actions of the middle layer.

Fig. 4 shows an example of the layered model, where the SUT was extremely simple (only 3 abstract and 3 concrete states). The management layer has information about the exact sequence generated by TESTAR. During the model inference, when TESTAR arrives to a new state and discovers actions that have not been executed before, we use “BlackHole” state as their destination to mark

Algorithm 1 ASM_*statemodel*: select an unvisited action

```
Require: s                                          ▷ The state the SUT is currently in
Require: State_Model                                ▷ The state model that is being inferred
Require: path                                       ▷ Path towards an unvisited action
 1: if path ≠ empty then                            ▷ ASM is following a previously determined path
 2:     a ← path.pop()                              ▷ Selected action is next one on the path
 3: else        ▷ If the path is empty, we will create a new one to an unvisited action
 4:     reachableStates = getReachableStatesWithBFS(s, State_Model )
 5:     unvisitedActions ← empty
 6:     while (unvisitedActions == empty ∧ reachableStates ≠ empty) do
 7:         s' = reachableStates.pop()
 8:         unvisitedActions ← getActions(State_Model, s', unvisited)
 9:     end while
10:     if unvisitedActions ≠ empty then            ▷ Unvisited actions found with BFS
11:         ua ← selectRandom(unvisitedActions)     ▷ Randomly select an unvisited
12:         path ← pathToAction(ua)                 ▷ Calculate path from s to walk to ua
13:         a ← path.pop()          ▷ Selected action in s is next one on the path to ua
14:     else                                        ▷ No unvisited action found with BFS
15:         availableActions ← getActions(State_Model, s, all)   ▷ Get all actions in s
16:         a ← selectRandom(availableActions)
17:     end if
18: end if
19: return a
```

unvisited actions. When a previously unvisited action is visited and TESTAR observes the SUT behaviour, the destination of the executed abstract action is updated with the observed abstract state.

4 Using the inferred models for action selection

TESTAR was extended with a new ASM (ASM_*statemodel*). The algorithm prioritises actions that have not yet been visited and can be found in Algorithm 1. The goal is to select a new action when in state s. It uses the *State_Model* and maintains a *path* of actions that leads to a specific unvisited action it wants to prioritise. If a *path* has been previously identified (i.e., *path* is not empty, line 1), the ASM just selects the next action on that path. If the *path* is empty, the ASM will try to find an unvisited action. It does so by searching (in BFS order) for *unvisitedActions* (line 8) from all the states that are reachable from s in the state model (line 4). Since s is reachable from s in 0 steps, s itself is the first state that gets checked for unvisited actions (line 7). If unvisited actions are found, it randomly selects one (*ua*, line 11) and updates the *path* to the state where that action can be found (line 12). Then it selects the first action that leads towards that action (line 13). If no unvisited actions are found, the ASM just randomly selects an action from those available in state s.

Table 1: Java Access Bridge properties and the possible impact of using the attribute for state abstraction in Rachota

Attribute	API	Abstract representation impact
Title	name	Visual name of the widget. In Rachota this is a dynamic attribute because widgets update the current time.
HelpText	description	Tooltip help text of the widget. In Rachota this attribute is static.
ControlType	role	Role of the widget. Cannot always distinguish different elements, and hence can cause non-determinism.
IsEnabled	states	Checks if the widget is enabled or disabled.
Boundary	rect	Pixel coordinates of the widgets' position. Even one pixel change would result in a new state, so it was considered too concrete for the experiments.
Path	childrenCount + parentIndex	Position in the widget-tree. Useful for differentiating states based on the structure of the widget tree.

To measure the performance of the new ASM_*statemodel*, we research the following question: *How do different levels of abstraction affect the automated GUI exploration of ASM_statemodel as compared to random (ASM_random)?*

GUI exploration is measured in terms of code coverage (instruction and branch) and the number of discovered states. Code coverage is measured using JaCoCo [1]. These metrics are collected after each executed action. To be able to measure code coverage, we use open source Rachota [2] as our SUT. It is a Java Swing application for timetracking different projects. It has the following characteristics:

Java Classes	52
Methods	934
LLOC	2722
Classes incl. Inner classes	327

Each test run contains one sequence of 300 actions, which is enough [10] to show the differences between ASMs. To be able to form valid conclusions and to deal with the randomness, we repeat each test run 30 times [9].

To define different abstraction levels, we need to select attributes from the available ones. In Table 1 are the widgets attributes from the Java Access Bridge API that are implemented in TESTAR for Java Swing applications. We investigated 4 levels of abstraction:

1. `Abstract`: ControlType (cf. was defined in [13])
2. `Intermediate`: ControlType, Path
3. `Dynamic`: ControlType, Path, Title (including the dynamic attribute Title)
4. `Customised Abstraction`: ControlType, Path, HelpText, IsEnabled (this one was customised for Rachota following the impacts described in Table 1)

The results of the code coverage measurements are in Fig. 5. They clearly show that the level of abstraction affects the GUI exploration performance of

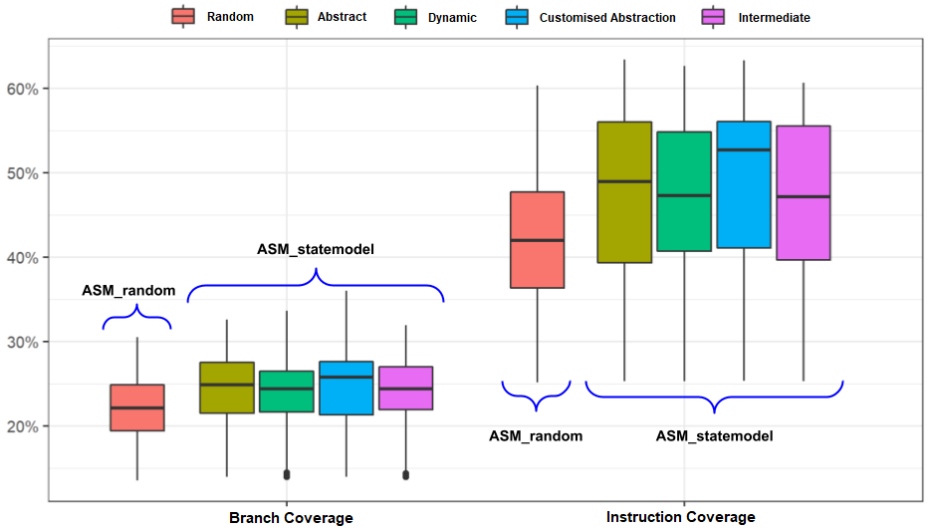


Fig. 5: The code coverage (%) that was reached when comparing ASM_*random* with 4 different abstraction levels of the ASM_*statemodel*

the ASM_*statemodel*. Having too high or too low level of abstraction negatively impacts the performance. However, the ASM_*statemodel* outperformed ASM_*random*, even with a less suitable abstraction. This means that model-based ASMs are a promising way to improve effectiveness of scriptless testing.

To investigate the impact of the changing levels of abstraction used in the experiments on the number of abstract and concrete states created, we run longer test runs of 3000 actions, again run 30 times for each configuration. The results are shown as a box plot in Fig. 6. Results show that too concrete level of abstraction creates almost as many abstract states as concrete states. As expected, the customised level creates more abstract states compared to abstract and intermediate configurations, but significantly less than the dynamic one. The customised level finds most concrete states, which indicates slightly better GUI exploration capability and matches the code coverage results.

5 Defining a suitable level of abstraction

The challenge is to select a suitable subset of widget attributes for state abstraction that does not cause state-explosion. Since non-determinism of the resulting model negatively impacts its usefulness for ASMs, we run experiments to research the following question: *Can we identify widget attributes that, when used in state abstraction, generate deterministic models?*

The SUT used is the desktop application "Notepad", specifically version 1909, OS Build 18363.535. We use Notepad because it is a Windows desktop application and hence we can use the Windows Automation API that gives us over 140 attributes to choose from. That gives more choice compared to the 6 at-

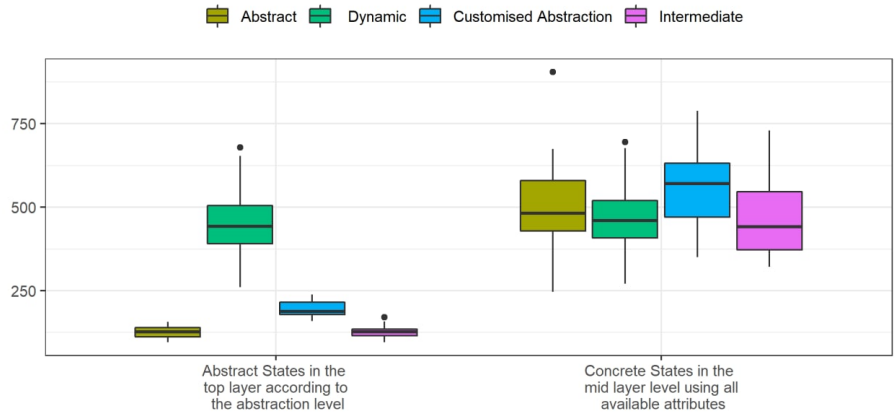


Fig. 6: The number of abstract states (top layer) and concrete states (mid layer)

tributes of the Java Bridge from Table 1. For testing, we use the ASM_*statemodel* from Algorithm 1.

We have seen in the previous section (and Figure 6) that widget attributes used for abstraction should not be dynamic because they lead to state space explosion. Dynamic attributes are not stable because they can change their value during, or in between, runs without a detectable reason. Potentially stable attributes selected for our experiment are in the first column of Table 2.

First, we run an experiment with only one attribute in the state abstraction. A test run consisted of 4 sequences, with a maximum of 50 actions per sequence. Running some initial tests, these values showed to be enough to detect non-determinism. We run 12 consecutive test runs for each widget attribute because we have 12 Virtual Machines (VMs) available. Table 2 shows the results. Displayed are the used widget attributes, the average number of generated test steps executed in the test for each attribute, and the total number of steps taken in each test run, before non-determinism was encountered. The results are ordered by the total number of steps executed over all 12 tests, starting with the widget attributes that "lasted the longest" before the model became non-deterministic. Although none of the models that were generated were deterministic, WidgetTitle, WidgetBoundary and WidgetHasKeyboardFocus attributes noticeably stand out from the other attributes by the average number of steps executed.

Second, we run an experiment with two attributes in the state abstraction. Combining 2 widget attributes, gives 171 possibilities. Instead of doing 4 test sequences of 50 steps each, we upgraded the number of actions per sequence to 100. The reason for the upgrade was the hypothesis that these combinations should last longer before the state model inference module encounters non-determinism. Each combination is tested 16 times, making for a total of 2736 test runs. In summary, none of the 171 combinations was able to produce a deterministic model. Moreover, after the 48 best performing combinations, the average number of steps executed per test run declines quickly. Within these 48 combinations,

Table 2: Number of generated test steps before the model became non-deterministic using a single widget attribute for state abstraction.

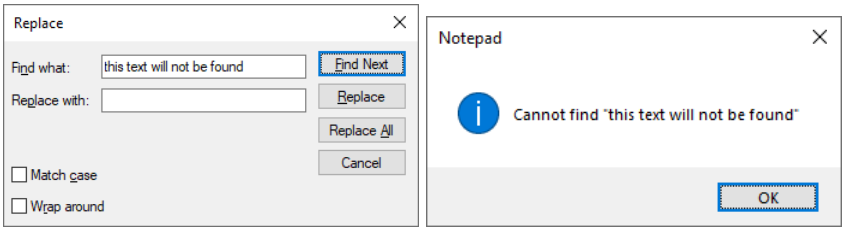
Attribute	**Avg**	**Total**
WidgetTitle	95.5	14,74,74,79,79,92,92,93,108,136,145,160
WidgetBoundary	90.6	5,61,64,77,79,83,84,103,105,115,155,156
WidgetHasKeyboardFocus	82.1	38,55,67,68,70,72,78,87,100,105,118,128
WidgetIsKeyboardFocusable	21.6	9,12,14,16,17,17,19,20,26,28,28,53
WidgetSetPosition	20.4	10,11,12,12,13,16,18,21,21,22,26,63
WidgetIsContentElement	20.3	7,9,14,15,17,17,18,21,24,27,35,39
WidgetIsOffscreen	19.7	6,9,11,14,14,15,15,18,19,27,29,59
WidgetGroupLevel	19.1	7,10,11,11,12,13,15,18,22,29,33,48
WidgetClassName	19.0	11,11,15,16,17,19,19,19,21,22,26,32
WidgetIsControlElement	16.9	8,11,11,12,13,13,16,16,20,24,28,31
WidgetIsEnabled	16.8	7,8,14,15,15,16,16,19,19,20,25,28
WidgetControlType	16.3	8,13,13,13,13,13,17,17,18,19,25,27
WidgetOrientationId	16.2	8,9,12,13,16,16,17,19,19,21,21,23
WidgetIsPassword	15.8	6,10,10,12,14,14,17,17,19,20,22,28
WidgetZIndex	15.6	9,11,12,12,13,14,15,16,16,19,22,28
WidgetIsPeripheral	15.4	7,8,9,13,14,14,16,19,20,21,22,22
WidgetSetSize	15.0	7,9,10,12,14,15,16,17,17,18,21,24
WidgetFrameworkId	15.0	8,11,12,13,13,14,15,16,17,18,21,22
WidgetRotation	14.7	8,9,12,12,13,14,16,16,16,20,20,20

we find that there are 3 attributes that occur 17 times, where as the next best attribute occurs only 3 times. The 3 best performing attributes are again: WidgetTitle, WidgetBoundary and WidgetHasKeyboardFocus.

For our next experiment, we used the combination of these 3 and run this combination 16 times. Unfortunately, none of the test runs made it to the full 400 possible steps. Moreover, the average and median are lower than the highest results from the 2 attribute combinations.

In our next experiment, we tried using combinations of 5 attributes, selecting the 3 highest scoring attributes from the 2 attribute experiment again, and then adding on all the possible combinations of 2 widget attributes from the other 16 attributes. This gives us a total of 120 combinations and we run each of them 8 times. Again, no combination resulted in a deterministic model. Surprisingly, the more concrete abstraction using five attributes resulted non-determinism faster than three attributes in the previous experiment. This is probably due to dynamic nature of some of the attributes.

As the use of 5 attributes for abstraction also resulted in non-determinism, we opt to make the model even more concrete by incorporating all 32 of the control pattern properties into our tests. To make some headway, we will take the 3 high scoring general properties again (WidgetTitle, WidgetHasKeyBoardFocus and WidgetBoundary), and combine them with all the combinations of 2 control patterns. This results in 492 possible combinations, and running each one 8 times makes a total of 3936 test runs.



(a) Notepad 'Replace' dialog (b) Notepad 'Cannot find' popup

Fig. 7: Notepad examples of non-determinism

Several widget combinations were able to reach the limit of 400 sequence actions without encountering non-determinism in the model, and all of these combinations included the 'Value' control pattern. Even though some combinations made it to 400 sequence steps 3 or 4 times out of the 8 test runs, they also encountered certain actions that led to non-determinism in the model. 'Value' pattern is a very 'concrete' attribute: 1) Using the 'Value' pattern can lead to models of infinite size, in the case that the application accepts text input that is not bounded. Hence, ideally, we would like to not use it in our abstraction mechanism. 2) Even while using this very concrete widget attribute, we still encountered plenty of non-determinism in the state models.

Analysing the reasons for non-determinism, we found actions that lead to different states depending on the history of actions and states traversed before. For example, in Notepad, if the 'Replace' option in the 'Edit' menu is clicked, the 'Replace' dialog is opened (see Fig. 7a). If the text written in the 'Find what' field is not found in the Notepad document, clicking 'Find Next' or 'Replace' buttons will result in the same popup dialog (see Fig. 7b), having only an 'OK' button. Clicking that button will lead back to 'Replace' dialog, but the focus remains on the button that was pressed before, and if WidgetHasKeyBoardFocus was used in the state abstraction, clicking the 'OK' button leads to 2 different states based on the action that was taken in the previous state. In this case, altering the abstraction level by adding more widget attributes would not remove the non-determinism, because the concrete states for the 2 visitations of the popup screen are also the same.

5.1 Including the predecessor state

Another solution we can try is to incorporate the state's incoming action into our state identifier [6]. In some situations, the state could depend on the previous state, requiring taking the previous state into account in the state identification algorithm. Consequently, we decided to include the predecessor state and the incoming action in the state abstraction.

The first experiments run with all the combinations of widget attributes used in the experiments from Section 5 including the previous state identifier. Non-determinism related to viewing the status bar was still happening.

Subsequently, we included the incoming action in addition to the previous state in the abstraction identifier. Using the same attributes as the experiments in Section 5 allowed for comparison of the results. The average number of steps executed before encountering non-determinism increased significantly when using 1 or 2 widget attributes and including the incoming action. However, with more widget attributes, the results seemed to get worse, probably because in those experiments the widget attributes were selected for their good performance without incoming action. With incoming action, the best performing widget attributes were different. When executing the experiments including pattern attributes with incoming action, the combination of the ValuePattern and the 'incoming action' seems very successful. In fact, the following 3 combinations did not encounter any non-determinism during their 8 test runs of 400 actions:

1. Boundary, HasKeyboardFocus, LegacyIAccessiblePattern, Title, ValuePattern;
2. Boundary, DropTargetPattern, HasKeyboardFocus, Title, ValuePattern;
3. Boundary, ExpandCollapsePattern, HasKeyboardFocus, Title, ValuePattern.

As some of the detected cases of non-determinism were due to various length of text inputs, we run an additional experiment disabling input actions, only allowing left click actions. We found that the average number of executed steps before encountering non-determinism was increased. However, the model may be partial, and some functionality of the SUT may be excluded from the model.

After running more experiments on trying to produce a deterministic model, we had to conclude that it is not a trivial effort. Also, the quest for inferring a deterministic model by making the abstraction more concrete resulted with huge increase in the number of abstract states. We discuss the implications on possible future research directions in Chapter 6.

6 Findings, challenges and future research directions

6.1 State abstraction

Our first finding is that tuning the abstraction level for model inference seems to be highly dependent on the specific SUT. While tuning the abstraction level, the following SUT-specific characteristics should be taken into account:

- **Dynamic increment of widgets**: In some applications, for example Rachota, we can find dynamic lists of elements on which we can constantly add new items. This constantly creates new widgets and states in the model causing a state and action explosion.
- **High number of combinatorial elements**: In some applications, for example Notepad, we can find multiple scroll lists with a large number of different elements, and from a functional point of view it is not important to cover all of these options (for example, Notepad Font selection).
- **Slide actions**: In some applications where scrolling actions are required, the exact scrolling coordinates from start to end can cause a change in the number of widgets visible in the state. Depending on the state abstraction and how the widget tree is obtained, this can create new states and cause a high number of combinatorial possibilities.

– **Popup information**: In some applications, for example Rachota, a descriptive popup message may appear for a few seconds in the GUI when the mouse is hovered over some of the widgets. This could result in a new state for the model, and it might cause non-determinism, if the hovering over a widget was not an intentional action that was executed on purpose.

A second finding is that trying to produce a deterministic state model is far from a trivial effort. There are various options to address the challenges of non-determinism in the inferred models. 1) The first could be to let the models have non-determinism and deal with it when using them. For action selection this means that we have to be able to detect when the modelled behaviour differs from the observed behaviour, and temporarily change the action selection to prevent getting stuck during GUI exploration. Another solution, and an interesting future research topic, could be using the concrete state model to navigate through states that have non-determinism in the abstract state model. 2) The second option would be to try to infer a deterministic model. This would require more SUT-specific ways to dynamically adjust the abstraction, for example based on the type of the widget, or even based on a specific widget in a specific state. TESTAR currently allows triggered behaviour that overrides normal action selection, so we plan to implement a similar way to trigger change in the calculation of state identifiers, for example ignoring a specific widget during the state abstraction. An example of a widget to be ignored from the state model could be a dynamically changing advertisement in a website. 3) A third option could be to correct non-deterministic models after run-time. Nevertheless, we have not seen this kind of technique used in a model-based testing tool yet.

Other interesting future research directions include automatically adjusting the level of abstraction, analysing the screenshots in addition to the attributes of the widget tree, and/or visualising the results of state abstraction for the user and learning from the user input to find a suitable level of abstraction.

6.2 Applying the inferred models in testing

One of the core objectives for this work was to use the inferred models for a new action selection mechanism (ASM) for TESTAR. The new ASM was presented in Algorithm 1 and initial experiments show that it is better than random. Although this is a good result by itself, it is also a step towards implementing more advanced ASMs. For example ASMs based on reinforcement learning (RL) and artificial intelligence (AI) need some kind of model for learning, and the inferred model can serve that purpose.

Another advantage of the inferred state models is that human testers can use them during testing. For instance, it is interesting to have an overall view of an application's execution flow: to see the details about a certain state or executed action; to identify the path to a state where an application failed; and to obtain various metrics about the state model. Although some of this information can be obtained by querying the OrientDB database and outputting it as textual data, e.g., in tabular format, we advocate that it would be best realised by visualising the data in a way that is more intuitively understandable for humans.

Abstract state models can also allow performing conformance testing. Conformance testing is used to determine how a system under test conforms to meet the individual requirements of a particular standard. Before using inferred abstract models, the domain experts have to validate them in order to use the automatically generated test cases for conformance testing. This also requires suitable visualisation.

Another future research topic is using the inferred models for automatically finding a shortest path to reproduce a failure. This requires recognising whether a found failure is a new one or duplicate of the one we try to reproduce. Finally, the inferred models can be used for automated change detection between consequent versions of the same SUT. This kind of functionality has been evaluated with Murphy tools [6], and it is interesting as future research.

7 Conclusions

This paper describes the state model inference extension for TESTAR and reports experiments on the impact of various state abstraction mechanisms for the purpose of producing a deterministic model, and on the evaluation of the performance of an action selection algorithm using the inferred models.

The experiments on using various state abstraction mechanisms show that inferring a deterministic abstract state model is difficult, especially when trying to prevent the state space explosion. Based on our experiences, and the fact that in the literature many approaches using inferred models for GUI exploration or testing do not explain the details about state abstraction, more research and new more flexible abstraction mechanisms are needed. Also, dealing with the non-determinism in the inferred models is an important future research direction.

Based on the experiments on the impact of various levels of state abstraction for the performance of an ASM using the inferred models, we can conclude that having a suitable level of abstraction improves the performance of GUI exploration measured in code coverage. Having a too abstract or too concrete model has a negative impact on the performance. However, in our experiments, the ASM_*statemodel* performed better than the ASM_*random* with all tested abstractions.

Finally, as an immediate future work, we plan to conduct experiments on SUTs from industry in order to demonstrate the effectiveness, efficiency and usability of the TESTAR tool with the inferred models proposed on this work.

Acknowledgements

This work has been funded by ITEA3 TESTOMAT and IVVES, and H2020 DECODER and iv4XR projects. Thanks to: Arjan Hommerson, Stijn de Gouw, Stefano Schivo

References

1. Jacoco coverage tool. https://www.jacoco.org/jacoco/, last accessed 17 Jan 2022
2. Rachota timetracker. http://rachota.sourceforge.net, last accessed: 17 Jan 2022
3. Aho, P., Alégroth, E., Oliveira, R.A., Vos, T.E.: Evolution of automated regression testing of software systems through the graphical user interface. In: 1st Int. Conf. on Advances in Computation, Communications and Services. pp. 16–21 (2016)
4. Aho, P., Kanstrén, T., Räty, T., Röning, J.: Automated extraction of GUI models for testing. Advances in Computers, vol. 95, pp. 49–112. Elsevier (2014)
5. Aho, P., Räty, T., Menz, N.: Dynamic reverse engineering of GUI models for testing. In: 2013 International Conference on Control, Decision and Information Technologies (CoDIT). pp. 441–447 (2013)
6. Aho, P., Suarez, M., Kanstren, T., Memon, A.M.: Murphy tools: Utilizing extracted GUI models for industrial software testing. In: 7th ICSTW. pp. 343–348 (2014)
7. Aho, P., Suarez, M., Memon, A., Kanstrén, T.: Making GUI testing practical: Bridging the gaps. In: 2015 12th International Conference on Information Technology-New Generations. pp. 439–444. IEEE (2015)
8. Amalfitano, D., Fasolino, A.R., Tramontana, P., Amatucci, N.: Considering context events in event-based testing of mobile applications. In: 6th ICSTW. pp. 126–133 (2013)
9. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 33rd ICSE. p. 1–10. ACM (2011)
10. van der Brugge, A., Ricos, F.P., Aho, P., Marín, B., Vos, T.E.: Evaluating TESTAR's effectiveness through code coverage. In: Abrahão Gonzales, S. (ed.) XXV JISBD. SISTEDES (2021)
11. Canny, A., Palanque, P., Navarre, D.: Model-based testing of gui applications featuring dynamic instanciation of widgets. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 95–104 (2020). https://doi.org/10.1109/ICSTW50294.2020.00029
12. Couto, R., Ribeiro, A.N., Campos, J.C.: A patterns based reverse engineering approach for java source code. In: 35th IEEE Software Engineering Workshop. pp. 140–147 (2012)
13. de Gier, F., Kager, D., de Gouw, S., Tanja Vos, E.: Offline oracles for accessibility evaluation with the TESTAR tool. In: 13th RCIS. pp. 1–12 (2019)
14. Grechanik, M., Xie, Q., Fu, C.: Creating GUI testing tools using accessibility technologies. In: 2009 International Conference on Software Testing, Verification, and Validation Workshops. pp. 243–250. IEEE (2009)
15. Grilo, A.M., Paiva, A.C., Faria, J.P.: Reverse engineering of GUI models for testing. In: 5th ICIST. pp. 1–6. IEEE (2010)
16. Kervinen, A., Maunumaa, M., Pääkkönen, T., Katara, M.: Model-based testing through a gui. In: Grieskamp, W., Weise, C. (eds.) Formal Approaches to Software Testing. pp. 16–31. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
17. Kull, A.: Automatic GUI model generation: State of the art. In: 2012 IEEE 23rd ISSRE Workshops. pp. 207–212. IEEE (2012)
18. Marchetto, A., Tonella, P., Ricca, F.: State-based testing of ajax web applications. In: 2008 1st ICST. pp. 121–130 (2008)
19. Meinke, K., Walkinshaw, N.: Model-based testing and model inference. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. pp. 440–443. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

20. Memon, A.M., Banerjee, I., Nagarajan, A.: GUI ripping: reverse engineering of graphical user interfaces for testing. In: 10th WCRE. pp. 260–269 (2003)
21. Mesbah, A., van Deursen, A., Lenselink, S.: Crawling ajax-based web applications through dynamic analysis of user interface state changes. ACM Trans. Web **6**(1) (2012)
22. Miao, Y., Yang, X.: An FSM based GUI test automation model. In: 11th International Conference on Control Automation Robotics Vision. pp. 120–126 (2010)
23. Nedyalkova, S., Bernardino, J.: Open source capture and replay tools comparison. In: Proceedings of the International C* Conference on Computer Science and Software Engineering. pp. 117–119 (2013)
24. Nguyen, B.N., Robbins, B., Banerjee, I., Memon, A.: GUITAR: an innovative tool for automated testing of GUI-driven software. Automated software engineering **21**(1), 65–105 (2014)
25. Rafi, D.M., Moses, K.R.K., Petersen, K., Mäntylä, M.V.: Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: 2012 7th International Workshop on Automation of Software Test (AST). pp. 36–42. IEEE (2012)
26. Silva, J.a.C., Silva, C., Gonçalo, R.D., Saraiva, J.a., Campos, J.C.: The GUISurfer tool: Towards a language independent approach to reverse engineering GUI code. In: Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems. p. 181–186. ACM (2010)
27. Vos, T.E.J., Aho, P., Pastor Ricos, F., Rodriguez-Valdes, O., Mulders, A.: TESTAR – scriptless testing through graphical user interface. STVR **31**(3) (2021)