The Journal of Systems & Software 200 (2023) 111645

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Distributed state model inference for scriptless GUI testing☆

Fernando Pastor Ricós [a,*], Arend Slomp [b], Beatriz Marín [a], Pekka Aho [b], Tanja E.J. Vos [a,b]

[a] Universitat Politècnica de València, València, Spain
[b] Open Universiteit, The Netherlands

ARTICLE INFO

Article history:
Received 8 July 2022
Received in revised form 6 February 2023
Accepted 8 February 2023
Available online 15 February 2023

Dataset link: https://doi.org/10.5281/zenodo.7607233, https://nam11.safelinks.protection.outlook.com/?url=https%3A%2F%2Fdoi.org%2F10.5281%2Fzenodo.7607233&data=05%7C01%7Cs.ba%40elsevier.com%7C423c48cc07b542ea1fc008db0b4f83a1%7C9274ee3f94254109a27f9fb15c10675d%7C0%7C0%7C638116208362076629%7CUnknown%7CTWFpbGZsb3d8eyJWIjoiMC4wLjAwMDAiLCJQIjoiV2luMzIiLCJBTiI6Ik1haWwiLCJXVCI6Mn0%3D%7C3000%7C%7C%7C&sdata=PN3sXtmgvd9724qJJVzGWt0Z%2BpoL5a%2BrY1hxBbhS9yU%3D&reserved=0

Keywords:
Distributed systems
Scriptless testing
State model inference
Empirical evaluation

ABSTRACT

State model inference of software applications through the Graphical User Interface (GUI) is a technique that identifies GUI states and transitions, and maps them into a model. Scriptless GUI testing tools can benefit substantially from the availability of these state models, for example, to improve the exploration, or have sophisticated test oracles. However, inferring models for large systems requires a long execution time. Our goal is to improve the speed of the state model inference process. To achieve this goal, this paper presents a *distributed* state model inference approach with an open source scriptless GUI testing tool. Moreover, in order to be able to infer a suitable model, we design a set of strategies to deal with abstraction challenges and to distinguish GUI states and transitions in the model. To validate it, we conduct an experiment with two open source web applications that have been tested with the distributed architecture using one to six Docker containers sharing the same state model. With the obtained results, we can conclude that it is feasible to infer a model with a distributed approach and that using the distributed approach reduces the time required for inferring a state model.

© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

A Graphical User Interface (GUI) allows end-users to interact with a software application. Therefore, testing applications at the GUI level is essential because it tests from the end users' perspective. Automating test execution through the GUI can be categorized into scripted and scriptless approaches. **Scripted approaches** require scripts that define sequences of user actions to test the System Under Test (SUT). These scripts can be recorded, manually crafted, or generated from manually created models (Rafi et al., 2012). The scripts are created prior to test execution and if the SUT changes, the scripts have to be updated. **Scriptless approaches** automatically generate sequences of user actions on-the-fly at run-time to explore the SUT by selecting and executing the available actions in the discovered GUI states (Wetzlmaier et al., 2016; Haoyin, 2017; Aho and Vos, 2018). In contrast to scripted tools, the decision-making performed by scriptless tools to select which action to execute is not done following previously created or recorded scripts or models. The scriptless tools follow an action selection algorithm to decide what to execute next at run time.

Scriptless approaches can benefit from inferring a state model for different purposes, e.g., more intelligent action selection decisions (Mariani et al., 2014), defining offline oracles (de Gier et al., 2019), comparing models of different versions of a SUT to detect GUI changes (Aho et al., 2014), or for regression testing (Grilo et al., 2010; Aho et al., 2016). State model inference techniques identify GUI states s and the actions a that are executable in state s, and use those to infer state-transitions $s \rightarrow a \rightarrow s'$ from state s to state s' when a is executed in s.

Although inferring a state model for all these purposes has many benefits, unfortunately for large and complex SUTs it requires a lot of time. Moreover, considering agile development processes in the industry (Kropp et al., 2018), multiple development teams update different applications several times a day. Because these teams need results quickly, the inferred model

☆ Editor: A. Bertolino.
* Corresponding author.
E-mail address: fpastor@pros.upv.es (F. Pastor Ricós).

https://doi.org/10.1016/j.jss.2023.111645
0164-1212/© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

can become obsolete overnight. This paper proposes a distributed architecture that allows multiple scriptless testing tool instances to infer a shared state model, making the inference process faster and hence more scalable.

Distributed computing uses architectures with multiple components to achieve a common goal. While the number of executed actions cannot be reduced for a single instance to infer a model with sufficient coverage, a distributed architecture can divide the execution of all actions amongst multiple instances to reduce the required time. The scalability in the model inference could improve the speed of the inference process.

In order to obtain significant evidence of this improvement, we have applied the distributed solution to two open source web applications in a controlled experiment. The results of our experiment indicate that a distributed architecture improves the speed of the state model inference process.

Therefore, the main contributions of this paper are:

- A distributed architecture that allows inferring a centralized model.
- A shared knowledge algorithm that helps multiple scriptless testing instances to be coordinated when inferring the model.
- A proposal to deal with non-determinism in the model.
- A set of strategies to deal with the dynamism in the SUT.
- An empirical experiment with two web applications showing that a distributed approach reduces the time required for inferring a state model.

This paper is structured as follows. Section 2 presents the related work, the characteristics of the open source TESTAR tool, and the main challenges related to model inference. Section 3 describes our distributed approach and algorithm proposal that allows all instances to be aware of what other instances are discovering and deal with non-deterministic models. Section 4 describes the empirical study that measures the code coverage and the size of the inferred model over time to see whether the distributed approach infers the model faster by adding more tool instances to explore the SUT. Section 5 shows the results obtained. Finally, Section 6 presents our conclusions and future work.

2. Related work

The field of automated GUI testing faces several challenges, like the maintenance of scripts, increasing the test coverage, or reducing the complexity of the configuration of the tools. These challenges need to be solved or mitigated to improve the GUI testing of complex applications (Aho et al., 2015; Aho and Vos, 2018; Nass et al., 2021). Regarding scriptless GUI testing and state model inference, there are additional challenges, such as the time required to infer and use the models, dealing with the dynamic aspects and non-determinism in the GUI of the applications, and state space explosion in the models. Also, many of the GUI testing tools support only a specific platform, e.g., Windows desktop applications (Mariani et al., 2014), Android apps (Wen et al., 2015), or web applications (Mesbah et al., 2012).

Some tools, such as the PATS framework (Wen et al., 2015), use GUI ripping (Memon et al., 2013) to extract GUI information of Android applications to generate execution sequences on-the-fly. It uses a central controller that coordinates the distributed nodes to reach different parts of the SUT. They limit the execution time in order to deal with the state explosion provoked by finding infinite GUI states in some applications. In contrast to the ideas in this paper, PATS does not use the ripped information to build a state model. Furthermore, this research introduces the challenge of dynamism and a designed solution to deal with state explosion.

There are only a few distributed approaches for state model inference of GUI applications. Cartographer (Brach et al., 2011) proposes a distributed architecture with a central manager and parallel workers to infer a GUI model in a distributed way. Although they do not give details about the abstraction mechanisms used, they mention the importance of a suitable abstraction to infer a SUT graph. However, the paper only describes the approach briefly, using an example of Notepad as a SUT, does not mention GUI challenges other than an extensive state graph space, and does not contain any empirical evaluation.

Crawljax (Roest et al., 2010; Mesbah et al., 2012) creates a graph of Document Object Model (DOM) states and click event transitions. To address state space explosion, they constrain the depth level, indicating a maximum number of states or a maximum crawling time. To deal with dynamism, they detect similar DOM states with a pattern comparator that allows filtering of dynamic elements and their properties. Then, to control some degree of non-determinism, they define crawl conditions that check whether a state should be visited. We use a similar solution of conditions to ignore dynamic widgets and their properties. Nonetheless, we consider that non-determinism can also be caused by the lack of information in the GUI. For this reason, we decided not to try to control non-determinism but to detect and adapt the GUI exploration algorithm. Crawljax concurrent experiment shows that increasing the number of distributed instances makes it possible to decrease up to 65% of the inference process. However, Crawljax only considers mouse clicks and hover actions, which results in very limited testing of an application from the user's perspective.

Since the existing open source tools for distributed GUI model inference show disadvantages, we evaluated the effort to extend other GUI model inference tools with support for distributed GUI exploration. AutoBlackTest (Mariani et al., 2014) creates an abstract behavioral model of GUI states and actions. To identify the states, they apply an abstraction function to every widget in the state that selects a subset of widget properties. This subset of properties is selected according to the widget type and ignores the properties they considered inconstant. This tool allows the execution of click and type actions and the combination of them. They briefly mention the existence of non-deterministic sequences without details about the possible impact on the action selection algorithm. Nevertheless, AutoBlackTest is not actively maintained and is restricted to desktop applications.

Murphy tool (Aho et al., 2013) creates a graph to model the behavior of the GUI application. It uses screenshot comparison and available GUI actions for defining states, abstracting for example data values. This research mention the tends to have a high number of GUI states without details regarding dynamism or non-determinism. Although it supports many kinds of GUI interactions and has been evaluated in a company, the Murphy tool has not been actively maintained for a long time.

TESTAR is a Java-based open source tool that, while testing, infers a GUI state model that is stored in the OrientDB graph database (Vos et al., 2021). The tool provides multiple APIs to allow interaction with different types of systems. Windows Automation API can be used for Windows desktop applications, Java access bridge for Java Swing applications, Selenium WebDriver for web pages, and Appium for mobile applications. We selected the TESTAR to research the distributed approach because it is open source, supports testing of desktop, web, and mobile applications, supports many kinds of GUI interactions, infers a GUI state model at run-time, has been evaluated in the industry (Bauersfeld et al., 2014; Chahim et al., 2020; Ricós et al., 2020; Aho et al., 2021), and continues being under active maintenance and development.

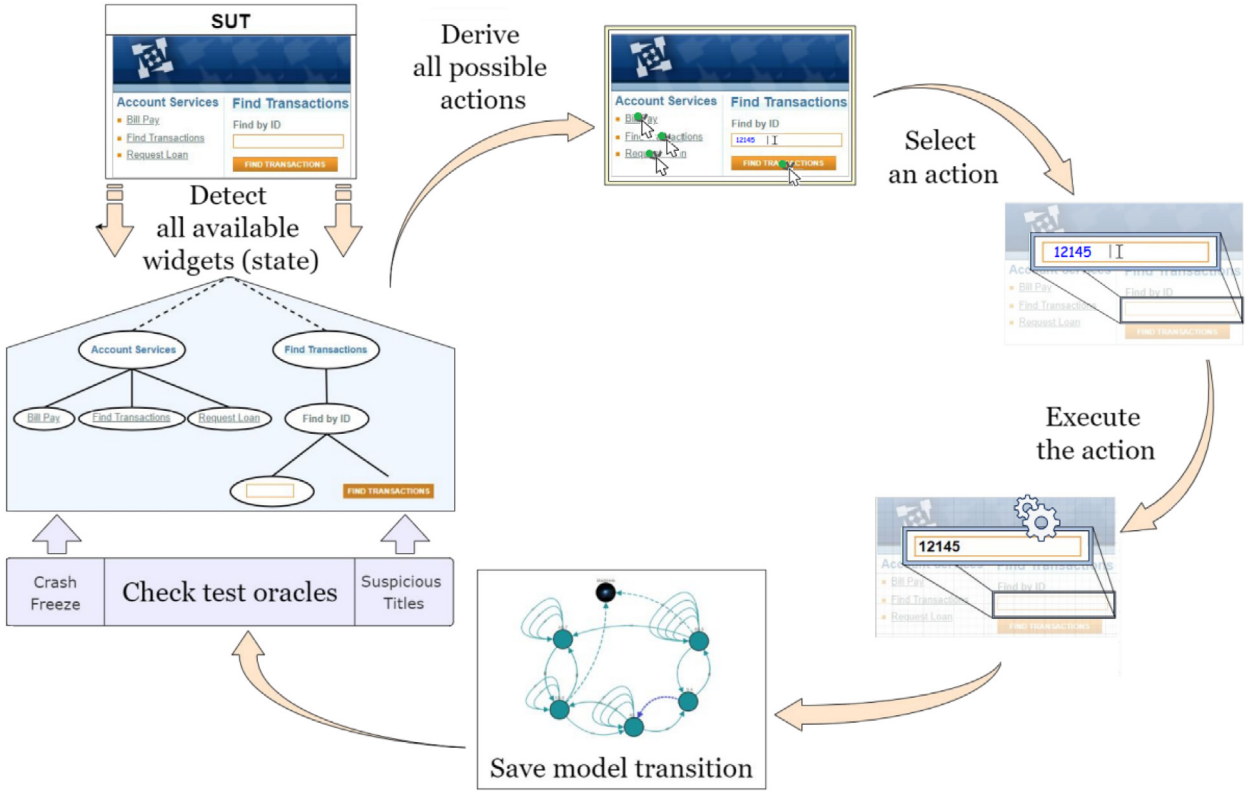

Fig. 1. TESTAR operational flow.

2.1. TESTAR tool

TESTAR connects to the SUT using one of the provided APIs to obtain the current GUI state. The GUI state is a hierarchical set of widgets, known as a widget tree. Based on some of the widgets' properties, the set of available actions, that TESTAR is able to execute, is derived automatically. For example, when a clickable widget button exists in the GUI state, TESTAR automatically derives an action that consists of a set of events: `MouseMove`, `MousePress`, and `MouseRelease`, that allows the tool to interact with the SUT.

Then, based on the available actions and the chosen action selection mechanism (ASM), TESTAR selects and executes an action a and obtains a new state s'. The transition (s, a, s') is then stored into the state model, and TESTAR continues with state s', generating test sequences until the STOP condition is met. The logical flow of TESTAR is shown in Fig. 1.

TESTAR stores concrete and abstract states in the model. Concrete GUI states consist of all the properties (that are available through the API) of all the widgets. Abstract GUI states consist of a selected subset of properties from all widgets. The subset can be configured by selecting which widget properties are used for state abstraction. The mapping between concrete and abstract states is surjective, meaning that each concrete state is mapped to one abstract state, whereas multiple concrete states can map to the same abstract state.

A similar approach is applied for GUI actions. For concrete GUI actions, the identifier consists of the concrete identifier of the state on which the action was executed, the coordinates of the interacted widget, the role of the action (i.e., click or type), and the specific text if the action includes typing. For abstract GUI actions, the identifier consists of the abstract identifier of the state, the hierarchical index of the interacted widget, and the role of the action without considering the typed text.

TESTAR offers an algorithm that uses the abstract information of the state model to prioritize the execution of actions that have been discovered but not yet visited (Mulders et al., 2022). However, as we explain in Section 3.2, this *state model algorithm* is not suitable for a distributed architecture.

2.2. State model inference challenges

Selecting an appropriate abstraction mechanism when trying to prevent the state space explosion is not a trivial task (Mulders et al., 2022). Whether the model inference is performed by a unique instance or with a distributed approach, the state model inference technique is negatively affected by two main abstraction challenges: dynamism and non-determinism (Nass et al., 2021).

Most SUTs contain states with dynamic widgets that can be created and removed as we explore the application, for example, bank account transactions or shopping cart elements. Because the default abstraction mechanism of TESTAR is not able to recognize and ignore dynamic widgets, the dynamic widgets can lead to constantly identifying new states in the model. This is called dynamism and can cause a state space explosion in the model and the constant increase of new actions (Memon, 2007; Clarke et al., 2011).

Another challenge is to deal with the non-determinism in the model (Roest et al., 2010; Luo et al., 2014). In a deterministic model, executing a specific action in a specific state always creates a transition to the same target state. However, a lack of information in the GUI not accessible with TESTAR APIs or a too generic abstraction strategy can result in an action that transits to a different target state when executed multiple times. For example, in an action that transfers money between two bank accounts, we can find non-determinism if the abstraction strategy ignores the amount of available funds and the account runs out of money.

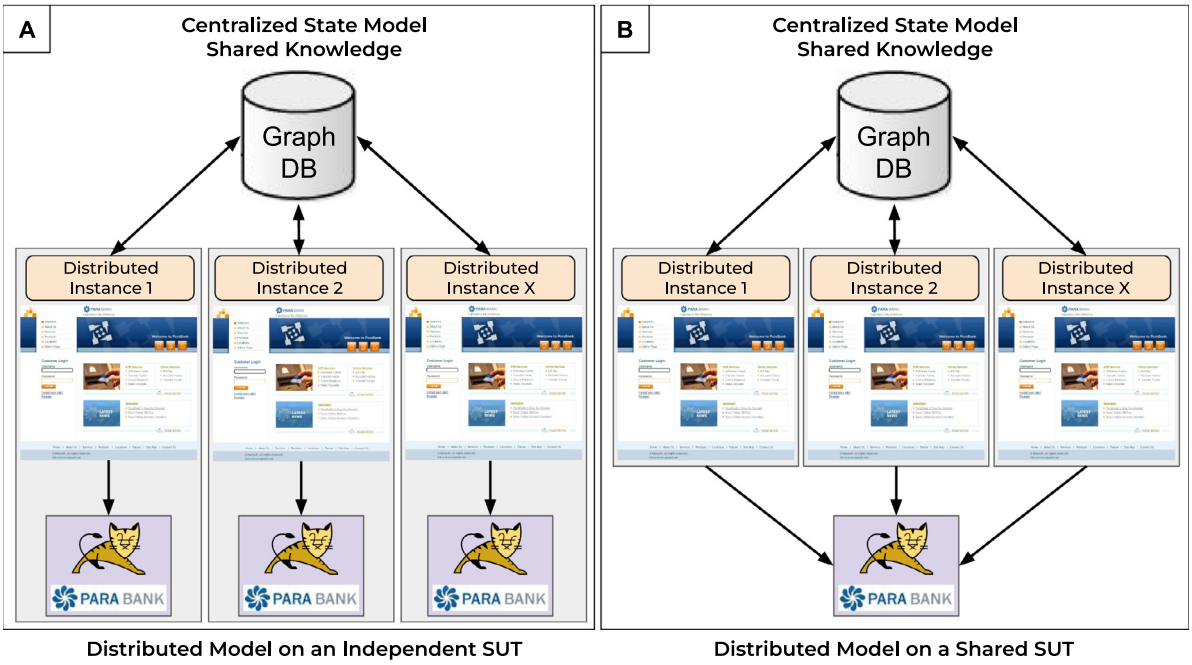

Fig. 2. Distributed state model inference architectures.

Moreover, dynamism and non-determinism challenges are dependent of each other. An abstraction strategy that ignores dynamic widgets can provoke non-determinism in the model. Section 3.3 provides more details about encountering non-determinism in complex applications and our proposed algorithm solution to deal with it. Section 4.4 indicates the designed abstraction sub-strategies used to deal with dynamic widgets.

3. Distributed state model inference

Distributed system architectures are implemented to let multiple agent instances achieve a common goal, in this paper, inferring a state model faster. To achieve this in a distributed manner, we will run multiple TESTAR instances at the same time, and the distributed knowledge they produce is joined together in a centralized and shared state model. At the same time, the TESTAR instances use this shared model to select actions and generate more distributed knowledge. The specific ASM designed to enable the instances to be aware of the information discovered by the others, called *SharedKnowledgeASM*.

The SUT, for which the model is inferred, can be deployed to manage independent connections (see Fig. 2A), or to manage multiple connections with shared data (see Fig. 2B).

Fig. 2A shows an architecture that we call Distributed Model on an Independent SUT (DMIS). This architecture could mitigate concurrent events, but we might find a lack of SUT information. For example, one distributed instance creates a user in an independent SUT instance. This user creation adds new functionality and new widgets in some states of the SUT. However, this new user only exists in that independent SUT, other SUTs have no users created. Then, because all distributed testing instances share the state model knowledge, they may think that new widgets are accessible if they navigate to some specific state. This may cause non-determinism in the shared model and the need to execute the same trace of actions in all the independent SUT to have the same states with the same data in all the distributed instances.

Fig. 2B shows an architecture that we call Distributed Model on a Shared SUT (DMSS). In this case, any distributed instance can alter the data being read by other instances. Therefore, in DMSS, concurrent events might affect the GUI information used to infer the model and provoke additional non-determinism challenges. However, in this case, the distributed instances would discover the new state with the common data. In this research, we focus on using this architecture because it represents most of today's web systems.

In the following subsections, we present the challenges addressed to allow TESTAR to infer a distributed state model and to facilitate its integration in distributed architectures.

3.1. Containerization of TESTAR and chromedriver

A distributed architecture requires a separate environment for each distributed instance. The industry is widely adopting containerization over virtualization because it provides efficient, scalable, and cost-effective resource management solutions in company infrastructures (Watada et al., 2019). A Docker container is a standalone unit of software that can package an environment, a SUT, and the GUI testing tool.

Containerization of a Windows environment with GUI software, however, is still a topic that requires further research. For desktop SUTs, multiple virtual machines have to be used. But, for web or mobile SUTs, it is possible to containerize the GUI environment. That is why we decided to first use Docker for distributed state model inference for web applications and, in the upcoming research, to continue with mobile and desktop applications.

The Selenium community made an Ubuntu based Docker image available with a standalone version of Selenium ChromeDriver (Selenium, 2022). This allows TESTAR to launch and interact with web SUTs. This Docker image also contains the *X virtual framebuffer* (Xvfb) (X.Org, 2022) server that simulates graphical operations without the need to have any graphical or screen device in the system environment. The use of this base image with the installation of the OpenJDK software allows the execution of a TESTAR tool Docker instance.

Algorithm 1 *SharedKnowledgeASM*

```
Require: s                                          ▷ The state the SUT is currently in
Require: StateModel                                 ▷ The state model that is being inferred
Require: targetAction                               ▷ UnvisitedAction marked as targetAction
 1: if targetAction == null then                    ▷ No UnvisitedAction has been marked yet
 2:     targetAction ← popClosestUnvisitedAction(s, StateModel)
 3: end if                                          ▷ At this point an UnvisitedAction is marked
 4: if s.contains(targetAction) then                ▷ If destination state reached
 5:     a ← targetAction                            ▷ Select UnvisitedAction marked
 6:     targetAction ← null                         ▷ And reset the mark
 7: else                                            ▷ Else, move one step closer to the destination state
 8:     shortestActions ← shortestPath(s, targetAction, StateModel)
 9:     a ← selectRandom(shortestActions)           ▷ Select one step action
10: end if
11: return a                                        ▷ Execute selected action
```

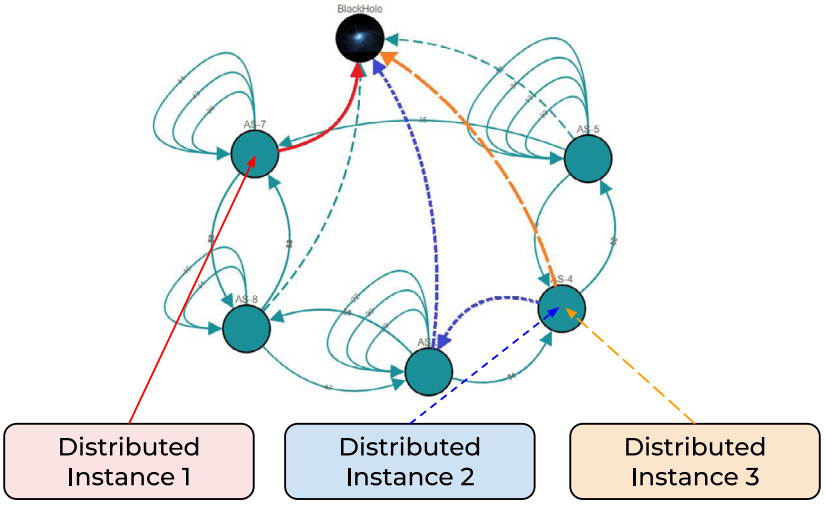

Fig. 3. Shared state model knowledge ASM.

3.2. Shared knowledge ASM

When running only one TESTAR instance, the *state model algorithm* maintains a previously identified shortest *path* of actions in the model that lead to a previously prioritized unvisited action (i.e. the *targetAction*) (Mulders et al., 2022). However, this approach will not work in a distributed setting because when the tool prioritizes the *targetAction* and selects the actions to get there, another, shorter path might become available through states discovered by other instances. Moreover, we need to prevent different instances from prioritizing the same *targetAction*. In a distributed architecture, all instances have to be aware of what other instances are discovering. The instances will do this by reading and writing a shared knowledge repository. To enable this, we developed the first version of a new ASM called *shared knowledge ASM*.

The shared knowledge ASM is used independently by each instance to re-calculate the shortest path to a targeted unvisited action every time a new action is selected. This means that each instance does not maintain a path of actions to reach the state that contains the *targetAction*, but only the *targetAction* that exists in the shared model. Moreover, because there is not a centralized component to dispatch the actions to coordinate all the instances, each independent instance has to mark the selected *targetAction* in the shared model to make sure that other instances do not pick the same *targetAction*.

Algorithm 1 presents the shared knowledge ASM for one TESTAR instance. If the TESTAR instance has no *targetAction* (line 1), the algorithm tries to find a new unvisited action closest to the current state *s*, and mark one of them as *targetAction* in the shared model to prevent other instances from selecting it (line 2). Suppose two different instances try to mark the same *targetAction* at a specific instant of time in the shared model. In that case, OrientDB manages the concurrent events. One of the instances receives an indication to wait a short time before trying to find another unvisited action closest to the current state *s* and mark them as *targetAction*.

At this point, the TESTAR instance already has marked a *targetAction*. To find the next action to select, the algorithm checks whether the current state *s* contains the *targetAction* action (line 4). If it does, the algorithm only needs to select the *targetAction* next (line 5) to reach its goal and reset the *targetAction* (line 6), such that the next iteration will search for a new *targetAction* (line 1). If the current state *s* does not contain the *targetAction* action (line 7), the algorithm calculates the actions that follow the shortest path from *s* to the target state that contains the *targetAction* (line 8). Because there may be several actions that lead to the shortest path, the algorithm selects one of them randomly (line 9). Finally, the algorithm returns the selected action (line 10).

Fig. 3 shows an example of how multiple distributed instances share knowledge of the state model. The nodes are the discovered states, the transitions between the nodes are the executed actions, and the black hole marks all the actions that were detected but remain unvisited. While *Instance 1* remains currently alone in a state with an unvisited action to execute, *Instance 2* and *Instance 3* are currently both in the same state, and they need to know what the other instance is going to do. In this case, *Instance 3* has marked first the unvisited action as *targetAction*, then *Instance 2*

Table 1
Preliminary results of Shared Knowledge ASM.

Instances	Runs	States	Transitions	Avg time	Factor
1	10	11	170	1070 s	0%
2	10	11	170	652 s	39%
3	10	11	170	545 s	16, 5%
4	10	11	170	530 s	2, 7%
5	10	11	170	517 s	2, 4%
6	10	11	170	530 s	−2, 4%

does not detect that action as unvisited and has decided to mark as *targetAction* an unvisited action that exists one state away.

A preliminary experiment was executed to validate the use of the shared knowledge ASM by multiple Docker instances. To do that, we configured a small web application on which it is possible to infer a complete and deterministic GUI state model. Table 1 shows the time it took for the different number of instances to infer the state model. We executed groups from 1 to 6 TESTAR docker instances for 10 runs to obtain the results. A complete run inferred a state model that consist of 11 states and 170 transitions. The average time indicates the time it has taken for each group of instances to infer the whole model. And the factor is the percentage time reduced between the group of instances and the previous one.

We can see that more distributed instances reduce the time needed to infer the model up to a certain limit. From 1 to 2 and 2 to 3 instances, the inference time is reduced by 39% and 16,5%, respectively. However, from 3 to 5 instances, there is only a small degree of improvement in the speed of the model inference. Moreover, changing from 5 to 6 instances increases the time for this small web application. The instances that do not find an available unvisited action will wait a short time for other instances to enrich the model before trying again. Therefore, we assume that using many instances for small applications will increase the waiting time, instead of inferring the model faster.

3.3. Dealing with non-determinism

The existing previous shared knowledge algorithm worked as expected when exploring small SUTs on which it is possible to infer a deterministic model. However, when applied to real and complex systems, the algorithm was negatively affected by the previously mentioned abstraction challenges: dynamism and non-determinism.

Dealing with dynamism challenges seems feasible by designing abstraction sub-strategies that indicate to TESTAR which widgets or properties it needs to use in different GUI states. However, dealing with non-determinism challenges is not always possible. Sometimes, the required information to differentiate a non-deterministic action is not available through the GUI. Furthermore, the use of the distributed approach with a DMSS architecture increases the issues that can arise related to non-determinism, since the actions executed by other instances can alter the information of the GUI. This challenge cannot be solved only by designing abstraction sub-strategies (as is done in Section 4.4). In addition, a new algorithm is required to detect and deal with non-deterministic actions, to allow navigating without getting stuck in loops, and inferring a state model covering the reachable parts of the GUI.

To illustrate the challenge of inferring a non-deterministic model, and our solution to deal with it in the new algorithm, we refer to Fig. 4. Imagine that, at a certain instant of time, during the inference process, a distributed instance I_x discovers that executing action *S1toS2* in *S1* transits to state *S2(A)*. For example, an action to transfer a certain amount of money between bank accounts. Now, suppose that in Fig. 4A another instance, I_y, is in state *S1* and uses the shared knowledge algorithm 1 to mark the *Unvisited Action* of *S2(A)* as the *targetAction* to execute. However, in Fig. 4B, after executing the action *S1toS2* with the intention to move to state *S2(A)*, I_y realizes it is not in the expected state. Instead of state *S2(A)*, it is in a new state *S2(B)* with a new set of *Unvisited Actions* to execute. This could happen, for example, when the transferring account runs out of money. Using algorithm 1, instance I_y still has the *Unvisited Action* marked in state *S2(A)*, but it will not be able to reach it, at least momentarily. This non-deterministic situation may cause a loop in the inference process.

To avoid such loops, we propose a solution that adds additional functionality to the TESTAR flow as shown in Fig. 1. After a transition is saved in the state model, it is checked whether the SUT reached the expected state. When this is not the case, action *S1toS2* is found to be non-deterministic, and hence all *S1toS2* actions are changed to point to a *Non-Deterministic Behavior* node (see Fig. 4C), meaning they cannot be used for navigating to states *S2(A)* or *S2(B)*. The algorithm then continues to prioritize first those *Unvisited Actions* to which distributed instances can navigate in the model. When we finished executing all reachable *Unvisited Actions*, the non-deterministic actions are executed again to restore the model transitions. This is illustrated in Fig. 4D, where the non-deterministic action *S1toS2* that discovered the transition to state *S2(B)* (and that was previously removed in Fig. 4C) is restored, and the transition pointer to the *Non-Deterministic Behavior* is deleted. After that, we can detect new *Unvisited Actions* in state *S2(B)* and continue the exploration.

The new non-deterministic shared knowledge ASM is presented in Algorithm 2. A new variable, *expectedState*, is added to the operational flow of TESTAR (see Fig. 1). This variable is updated when the tool selects an action to execute. After that, it is used in the new Method M1 when executing the selected action to save the new transition in the model. This new variable enables the detection of non-determinism in the path that leads to the *targetAction*.

Algorithm 2 first prioritizes unvisited actions as *targetAction* (lines 1 to 3). If no unvisited actions are available, the instance has no *targetAction* (line 4). Then, the algorithm tries to find a new non-deterministic action closest to the current state *s*. The non-deterministic action is marked as *targetAction* in the state model (line 5) to prevent other instances from selecting it. To find the next action to select, the algorithm checks whether the current state *s* contains the *targetAction* action (line 7). If it does, the algorithm only needs to select the *targetAction* to reach the target (line 8), reset *targetAction*, such that the next iteration will search for a new *targetAction* (line 9), and reset the *expectedState* because we reached the *targetAction* without non-determinism issues (line 10). If the current state *s* does not contain the *targetAction* action (line 11), the algorithm calculates the actions that follow the shortest path from *s* to the target state that contains the *targetAction* (line 12). Because there may be several actions that lead to the shortest path, the algorithm selects one of them randomly (line 13). Here, we update *expectedState* variable with the state we should navigate if the transition is deterministic (line 14). Finally, the algorithm returns the selected action (line 16).

Following the TESTAR flow from Fig. 1, this selected action is executed which results in a state transition that is added to the model. This is the moment where the *expectedState* variable is used to detect non-determinism. This is shown in Method M1 in case the *expectedState* is defined.

If the state to which the model has transitioned after executing the selected action is not the *expectedState*, the action is non-deterministic because it leads to two different states (Method M1, line 1). In this case, the action is marked as non-deterministic in the model (Method M1, line 2). Subsequently, the *targetAction*

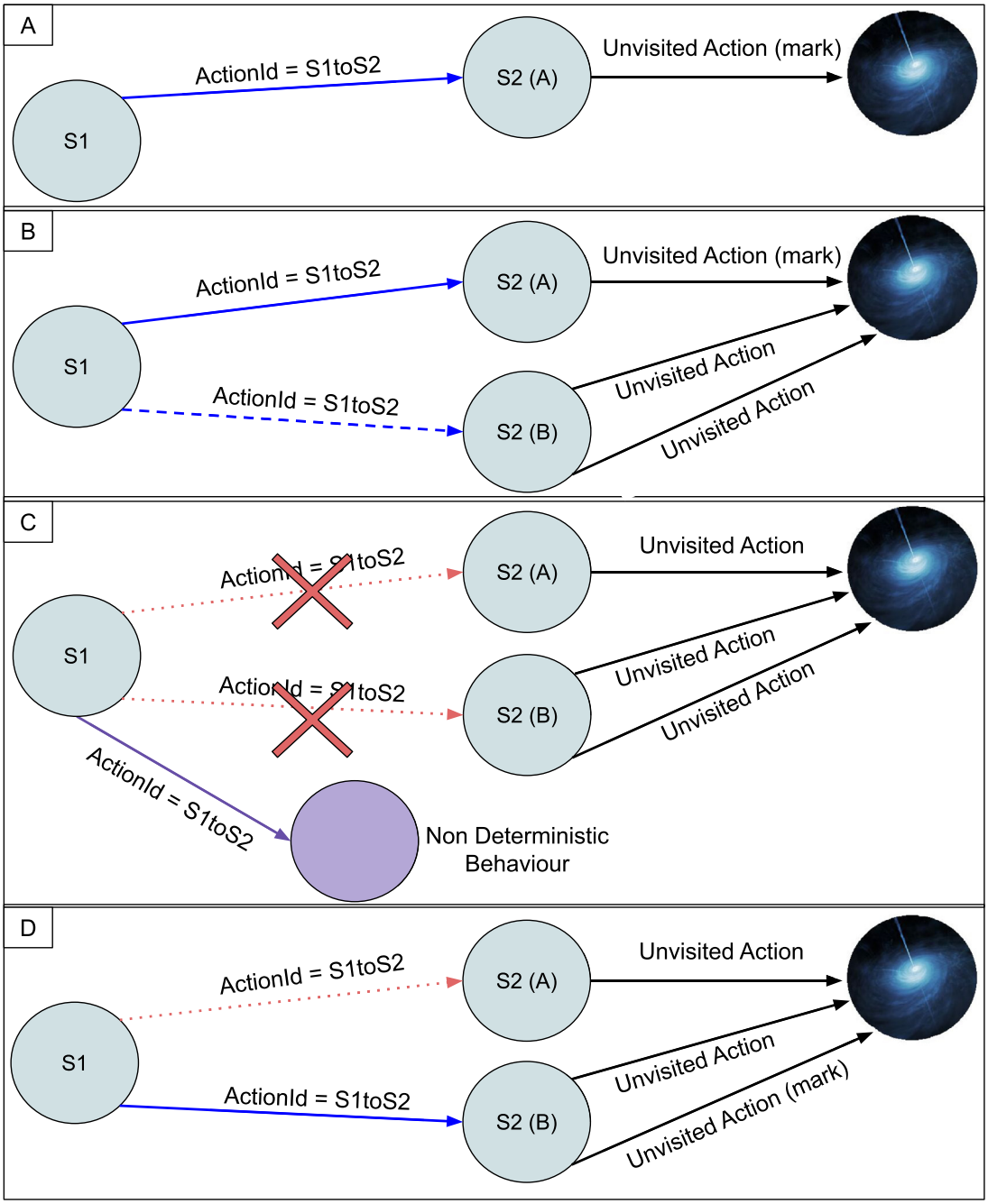

Fig. 4. Dealing with non-deterministic models.

is reset, so the next iteration will search for a new *targetAction* (Method M1, line 3). Finally, the *expectedState* variable is reset, because the next iteration will follow a new path (Method M1, line 4).

4. Empirical study design

The objective of the distributed implementation is to improve the speed of the state model inference process. We want to measure whether the distributed approach is more efficient in terms of the time needed to explore the SUT. To guide our study, we have formulated the following research question:

- `RQ1`: Does a distributed architecture allow scriptless testing tools to speed up state model inference?

To answer `RQ1`, we follow the guidelines suggested by Wohlin et al. (2012) to design a controlled experiment. This study follows a methodological framework specifically built to evaluate software testing techniques (Vos et al., 2012). Encouraged to disseminate the results, we favor choosing open-source systems for the experimentation. This experiment consists of a set of trials executing the distributed approach. In each trial, we measure variables related to the time efficiency of the SUT exploration and the state model inference process.

To make statistical analysis possible, we formulate the following null hypothesis based on the research question:

- H_0: The distributed approach does not reduce the time required to infer a state model.

Algorithm 2 *NonDeterministicSharedKnowledgeASM*

Require: *s* ▷ The state the SUT is currently in
Require: *StateModel* ▷ The state model that is being inferred
Require: *targetAction* ▷ UnvisitedAction or NonDeterministicAction marked as targetAction
Require: *expectedState* ▷ Expected next step state to navigate to deterministically
1: **if** *targetAction* == *null* **then** ▷ If no UnvisitedAction or NonDeterministicAction marked
2: *targetAction* ← *popClosestUnvisitedAction*(*s*, *StateModel*)
3: **end if**
4: **if** *targetAction* == *null* **then** ▷ At this point an UnvisitedAction is marked if available
5: *targetAction* ← *popClosestNonDeterministicAction*(*s*, *StateModel*) ▷ If no UnvisitedAction available in previous condition
6: **end if**
7: **if** *s.contains*(*targetAction*) **then** ▷ At this point a NonDeterministicAction is marked if available
8: *a* ← *targetAction* ▷ If destination state reached
9: *targetAction* ← *null* ▷ Select the UnvisitedAction or NonDeterministicAction marked
10: *expectedState* ← *null* ▷ Reset the target action mark
11: **else** ▷ Reset the determinism state check
12: *shortestActions* ← *shortestPath*(*s*, *targetAction*, *StateModel*) ▷ Else, move one step closer to the destination state
13: *a* ← *selectRandom*(*shortestActions*)
14: *expectedState* ← *nextDeterministicState*(*a*) ▷ Select one step action
15: **end if** ▷ Save the next step state we should navigate to deterministically
16: **return** *a* ▷ Return the selected action

Method M1 *Verify Action Transition Determinism*

Require: *expectedState* ! = *null* ▷ The expected state to navigate deterministically that was calculated in the Algorithm 2
1: **if** *currentState* ! = *expectedState* **then**
2: *markActionAsNonDeterministic*(*a*)
3: *targetAction* ← *null*
4: *expectedState* ← *null*
5: **end if**

Table 2
Details of the selected SUTs.

Metric	Parabank	Shopizer
SUT type	Web	Web
Java Classes	153	330
Methods	1127	2356
LLOC	4182	16 398
Instructions	18 288	74 926
Branches	1214	5258

4.1. SUT objects

The SUTs selected for this experiment, suitable for the methodological framework, must be published under an open-source license and should comply with the following:

(1) TESTAR is able to detect the GUI of the SUT;
(2) The SUTs can be deployed without a manual database configuration, which facilitates the automation and replication of the experiments;
(3) The SUTs are Java based. This allows using JaCoCo (JaCoCo, 2006, 2022), a well-known and widely used tool in the academic and industry sector, to obtain instruction and branch code coverage metrics for the empirical evaluation.

Based on these requirements, we selected the following SUTs:

- `Parabank` (Parasoft, 2017, 2022) is an open source online banking demo application that uses a Java implementation in the server side to process the management of customers, transactions, payments, and other bank services. This demo web application developed by Parasoft company is a representative SUT to test standard Java web services.
- `Shopizer` (CSTI, 2013, 2022) is an open source java e-commerce web app with two different domains: the administrator pages and the shop for customers. This web application is widely used by companies to offer product sales services. We leave out the admin section in these experiments and focus on the exploration of the shop.

We present the details of the selected SUTs in Table 2. The number of internal Java classes, methods, lines of code (LLOC), and instructions and branches indicate that these are not toy projects, i.e. they are representative SUTs of real applications. Moreover, these SUTs expose relevant GUI challenges, such as dynamism and non-determinism, that need to be faced in today's complex systems.

We needed to use the *blocking principle* (Wohlin et al., 2012) and disable the TESTAR oracles. This is because the usage of TESTAR oracles for fault detection involves stopping, reporting, and restarting the System Under Test (SUT). This feature affects the exploratory time used by the TESTAR instances, reducing the number of states discovered and actions executed. Using the *blocking principle*, we could focus the experiment only on the time required for the model inference process. The usage of oracles to detect failures will be investigated in future experiments. In order to prepare TESTAR for the experiment, we have configured a set of independent variables for each SUT that will be explained next.

4.2. Independent variables: time

A set of time variables are used to define the time budget of the experiments and the execution and waiting time of actions. It was necessary to run preliminary executions to determine an appropriate time budget for each SUT. Because Parabank is a web application formed mainly by a series of static GUI forms to use the bank services, 30 min was enough to reach a limit on the number of discovered states and decrease the unvisited actions. However, for Shopizer, it was necessary to increase the time budget until 240 min. This is because the web shop application is formed with a dynamic GUI that allows search, add, remove, and order products, creating a high amount of available states and action in the state model.

The time used to execute a GUI action is the same for both SUTs. Nevertheless, the time to wait between actions has been increased for Shopizer because some web states require more time to completely load the data in the GUI of some of the products. Table 3 shows the set of time variables used in our experiments for both SUTs.

Table 3
Independent time variables.

Variable	Parabank	Shopizer
Time budget	30 min	240 min
Action duration	1 s	1 s
Wait after action	1 s	2 s

Table 4
Independent variables for smarter action derivation.

Parabank	(Force) User login on each sequence (Force) Web history back on WSDL pages (Filter) Logout and admin panel buttons (Filter) Links outside parabank domain (Compound) Form to transfer a valid/invalid amount of money (Compound) Form to request a valid/invalid money loan (Compound) Form to find transactions by id, date or amount (Compound) Form to update customer information (Compound) Form for bill payment (Compound) Form for contact service
Shopizer	(Force) Accept cookies (Force) User login on each sequence (Filter) Logout and send mail buttons (Filter) Change language button (Custom) Actions in top level menus (Custom) Move mouse action for dropdown widgets (Custom) Only add product to cart if shopping cart is empty (Compound) Form to send an email to contact service (Compound) Form to change customer address (Compound) Form to change the account password (Compound) Form to realize the payment of the cart

4.3. Independent variables: derived actions

Regardless of the execution of a unique instance or the distributed approach, using the default set of derived actions that TESTAR realizes automatically is not enough to adequately simulate user interactions and effectively explore complex systems. In each SUT we need to derive smarter actions to maintain the exploration in the main GUI states or to reach some GUI states that require a specific combination of data or actions:

- *Force* actions are used to start and maintain the exploration in the desired state of the SUTs.
- *Filter* actions can be used to maintain the exploration in the main part of the SUT or save time by ignoring less essential features that may increase the exploration space.
- *Custom* actions are created for SUT specific widgets (e.g. top level menus, dropdown menus, shopping carts, etc.).
- *Compound* actions are added to the set of derived actions to deal with the filling of forms in one (compound) action.

Table 4 shows the derived smarter actions used in our experiments for Parabank and Shopizer.

Sliding actions offer an additional challenge when inferring a state model. These actions change the visible widgets in the GUI that can be interacted with, hence the abstract state. Depending on the SUT, the set of possible sliding positions can increase the exploration space. For this reason, and because it is possible to explore both SUTs without the need to execute sliding actions, we decided not to derive sliding actions.

4.4. Independent variables: abstraction strategy

TESTAR's default abstraction mechanism cannot differentiate type actions that type different texts, nor deal with the dynamism issues mentioned previously. Therefore, a new abstraction strategy that distinguishes a main mechanism and a set of sub-strategies, has been implemented in the tool by extending the default mechanism in the Java protocol of the tool.

Table 5
Independent variables for the main abstraction mechanism.

Parabank	
State	WebId, WebName, WebTextContent
Action	OriginState + OriginWidget + ActionRole
Compound action	OriginState + OriginWidget + ActionTypedText
Shopizer	
State	WebId, WebTextContent, WidgetIsOffscreen
Action	OriginState + OriginWidget + ActionRole
Compound action	OriginState + OriginWidget + ActionTypedText

Table 5 shows the main abstraction mechanism to identify the abstract states and actions of Parabank and Shopizer. Abstracting from dynamism can cause the same widget to change its hierarchical index in the widget tree of the abstract state. For this reason, instead of just using the widget tree hierarchical index for the abstract action, we use the origin widget abstract identifier. Compound actions now have a different abstraction from default actions because the content of the typed text is important when identifying an action transition in the state model.

Using only the main abstraction mechanism has limitations when applied in systems with dynamism. Each SUT may need the design of a set of abstraction sub-strategies to filter the dynamic widgets and their properties to avoid a state space explosion:

- Widgets that can be dynamically added or removed from the state, such as a dynamic table of elements or products of a shopping cart, are *ignored* when the tool generates the identifier of the abstract state.
- Widgets that contain properties with values that can change dynamically throughout the exploration, such as a dynamic text that displays user-editable information, need to use a *custom* set of specific properties to avoid using these dynamic values.
- Widgets that do not have enough default property values to differentiate them, such as dropdown widgets that do not contain a different web identifier, require new properties to be *added* to the abstraction mechanism.
- Widgets that respond to hover mouse events or states that require a longer time to load all web elements may provoke non-determinism in the model. Shopizer contains additional *events* intended to prevent this non-determinism.

Table 6 shows the design of abstraction sub-strategies for Parabank and for Shopizer.

4.5. Effort time for independent variables

Obtaining the right values for these independent variables requires user effort to identify, prepare and test. Most effort is needed to detect dynamic widgets and design the set of abstraction sub-strategies to implement the main abstraction mechanism for each SUT.

Parabank required approximately 8 h to prepare the smarter action derivation and 20 h to determine the best combination of properties for the main abstraction mechanism together with the design of abstraction sub-strategies. More than half of the time was used to test the implementation in the distributed docker architecture and verify the correct identification of states and actions in the model.

Shopizer, on the other hand, required approximately 16 h to prepare the smarter action derivation and 40 h to determine the best combination of properties for the main abstraction and the set of abstraction sub-strategies. This is a larger SUT regarding

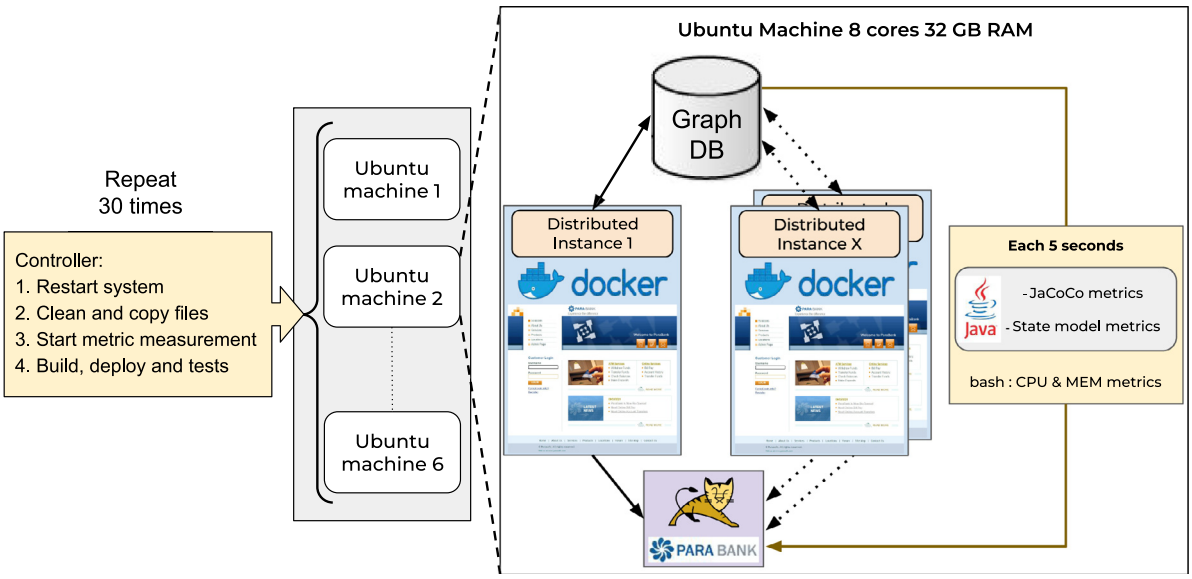

Fig. 5. Experiments execution.

Table 6
Independent variables for abstraction sub-strategies.

Parabank	(Ignored) Dynamic widgets from accounts table (Custom) Use only WebId for widgets of type select (Custom) Use only WebName for phone number widget (Custom) Use only WebId for new account widget (Custom) Use only WebId for widgets of bill payment state
Shopizer	(Ignored) Price and product widgets in the shopping cart (Ignored) Product and order number widgets in payment forms (Ignored) Address content in customer box info (Custom) Use only WebId in form widgets that dynamically change hint text. (Custom) Differentiate empty cart and cart with products (Added) Use parent id for edit bill and shipment widgets (Added) Use previous state id for Apache 404 state (Added) Use parent text content for dropdown widgets (Added) Use hyperlink reference for Home widgets (Event) Force mouse coordinates before each sequence (Event) Wait until loading overlay is complete

Table 7
Effort time to design the independent variables.

Time required for variable	Parabank	Shopizer
Smarter action derivation	8 h	16 h
Abstraction strategies	20 h	40 h

the number of discovered GUI states and actions which required more testing and verification time.

The time invested to learn and design smart actions and abstraction sub-strategies for a SUT may vary according to the user's expertise. It is important to note that the knowledge acquired helps to reduce the learning time in future configurations of similar SUTs. Table 7 contains an approximation of the times required by a TESTAR member to prepare the independent variables for Parabank and Shopizer.

4.6. Dependent variables

To answer RQ1, we need to measure whether, given a time budget, the distributed approach improves the speed of exploring and inferring a model. Thus, to compare the speed of the distributed approach we will measure the following:

(1) *Code coverage* over time to determine which parts of the code have been executed during the testing.
(2) *GUI coverage* over time to determine how many different GUI states and actions have been found in the SUT. For a fair comparison, we will use the same abstraction strategy for all the trials of the same SUT.

Since a distributed implementation requires more computation resources, we will also measure:

(3) *Consumption* metrics to measure the CPU and memory usage for each TESTAR Docker instance, the SUT, and the central graph database.

4.7. Design of the experiment

We compare the distributed approach with Parabank and Shopizer by running groups from 1 to 6 Docker testing instances, i.e., the first group with a single Docker testing instance, the second group with two distributed Docker testing instances, and so on, up to six distributed Docker testing instances. Each group of Dockers runs in an Ubuntu machine with 8 CPU cores and 32 GB RAM. All Docker containers are deployed with 512M shared memory size. To deal with the randomness and to be able to obtain valid conclusions about the possible rejection of the null hypotheses, we repeat the execution of the experiment 30 times. The SUT will start in the same initial status in each new execution. To do this, a bash controller takes care of:

(1) Restart the system to remove possible residual processes.
(2) Clean and copy Apache Tomcat with the web SUT, the graph database, and the TESTAR files; and prune the Docker containers of the previous executions.
(3) Launch a custom Java application and bash instructions to extract code coverage, state model GUI coverage, and consumption metrics every 5 s.
(4) Build and deploy Apache Tomcat with the web SUT, the graph database and a new number of Docker containers with TESTAR to start the testing process.

TESTAR logs and HTML output results, together with the coverage, state model, and performance metrics, are stored in a centralized data server at the end of each execution. Fig. 5 shows the overall design of the experiment.

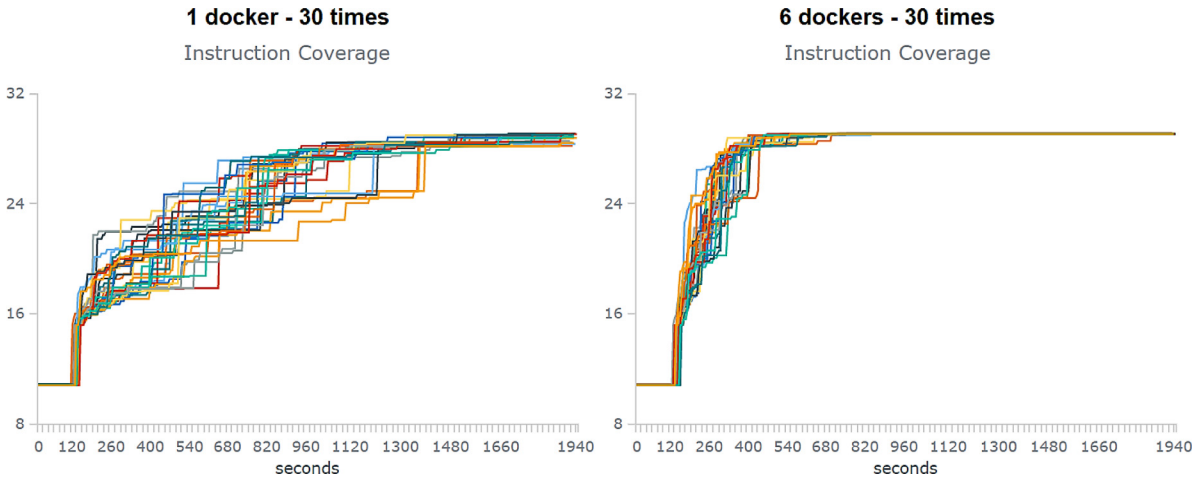

Fig. 6. Parabank code coverage comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.

5. Results

This section presents the results of the distributed approach for Parabank and Shopizer. Instruction code coverage is presented in curves comparing 1 with 6 groups of Dockers, and comparing the average code coverage of all Docker instances in a growth curve. We only show instruction coverage because branch coverage is visually similar. The graph curves for the six groups of Dockers are available online.[1] However, the subsequent statistical analysis is performed for both instructions and branch coverage. GUI coverage is presented in curves comparing 1 with 6 groups of Dockers. We use the Kruskal Wallis test for the instruction and branch coverage to check whether there is a significant difference between the six different Docker groups (Furia et al., 2019). Then, we use Wilcoxon and Cliff's delta tests to check which groups of Docker pairs differ significantly.

5.1. Parabank analysis

For a single Docker instance, the metrics obtained with JaCoCo indicates that the code coverage increases throughout the first 30 min of exploration (see Fig. 6). While, for six instances, the total code coverage is reached approximately in 10 min. For GUI coverage, we evidently observe a similar trend with the metrics extracted from the TESTAR state model (see Fig. 7). As new GUI abstract states are discovered in the TESTAR state model that represents the SUT, the code covered obtained with JaCoCo increases correspondingly.

Concerning the number of unvisited abstract actions for the GUI coverage (see Fig. 7), new unvisited actions are found during the first 20 min for a single Docker instance. Then, in the last 10 min, the number of unvisited actions starts decreasing. For six Docker instances, the peak of unvisited actions is reached in the first 5 min. After that, it decreases during 10 min, reaching a limit that does not go to zero. The latter is because some unvisited actions are unreachable due to non-determinism in the inferred models.

Fig. 8 represents the average code coverage of all Docker instances for Parabank in the first 700 s of execution. It is easy to see a growth difference between all the Docker groups except for 3 and 4 groups of Dockers.

We perform statistical analysis at 360, 480, and 600 s to obtain evidence about the significant difference in the distributed approach. This allows us to analyze how the distributed approach evolves from the moment the distributed instances have started exploring until the moment the groups with the most Dockers have reached the coverage limit. The Kruskal Wallis test indicates a significant difference between all distributed Docker groups in these three instants of time (see Table 8).

Table 8
Parabank Kruskal Wallis results.

Parabank	Instruction coverage		Branch coverage	
	H-value	P-value	H-value	P-value
360 s	115.11	3.39e−23	113.33	8.09e−23
480 s	130.17	2.19e−26	130.3	2.05e−26
600 s	147.18	5.32e−30	148.07	3.44e−30

At 360 s (see Table 9), Wilcoxon indicates no significant difference only for 3 & 4 groups of Dockers and a significant difference for the other groups of Dockers. Cliff's delta points negligent difference for 3 & 4 groups of Dockers, medium difference for 1 & 2, 3 & 5, and 4 & 5 groups of Dockers, and large difference for the other groups.

At 480 s (see Table 10), Wilcoxon continues indicating no significant difference for 3 & 4 groups of Dockers and a significant difference for the other groups of Dockers. Whereas Cliff's delta has changed and only indicates negligent difference for 3 & 4 groups of Dockers, the other Docker groups have large differences.

Finally, *at 600* s (see Table 11), Wilcoxon indicates that all Docker groups have significant difference, including 3 & 4 groups of Dockers. And Cliff's delta points that 3 & 4 and 4 & 5 Docker groups have small significant difference for instruction coverage, medium difference for 4 & 5 groups of Dockers regarding branch coverage, while the other groups of Dockers have large difference.

Related to Parabank computation resource consumption, Table 12 presents the CPU and memory consumption average of all groups of Dockers. CPU and memory consumption are increased around 6%–7% and 5%–6% respectively, for each new distributed Docker instance executed.

5.2. Shopizer analysis

Regarding Shopizer code coverage (see Fig. 9), a single Docker instance needs 2 h or more to reach the maximum coverage. While, with six Dockers, except for a couple of outliers, the maximum coverage is reached in approximately one hour of execution.

The maximum instruction coverage reached by the TESTAR instances in all groups of dockers was approximately 18.3%, except

[1] https://doi.org/10.5281/zenodo.7607233

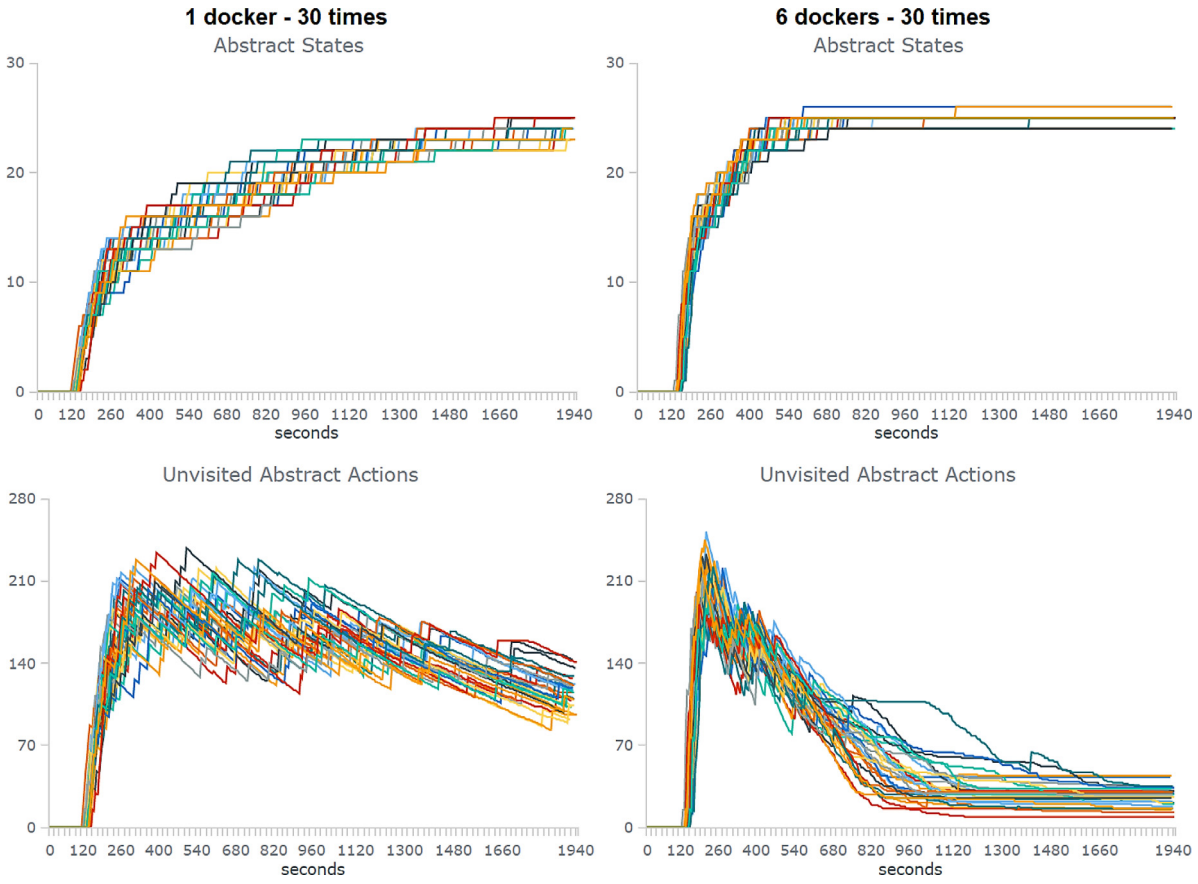

Fig. 7. Parabank state model comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.

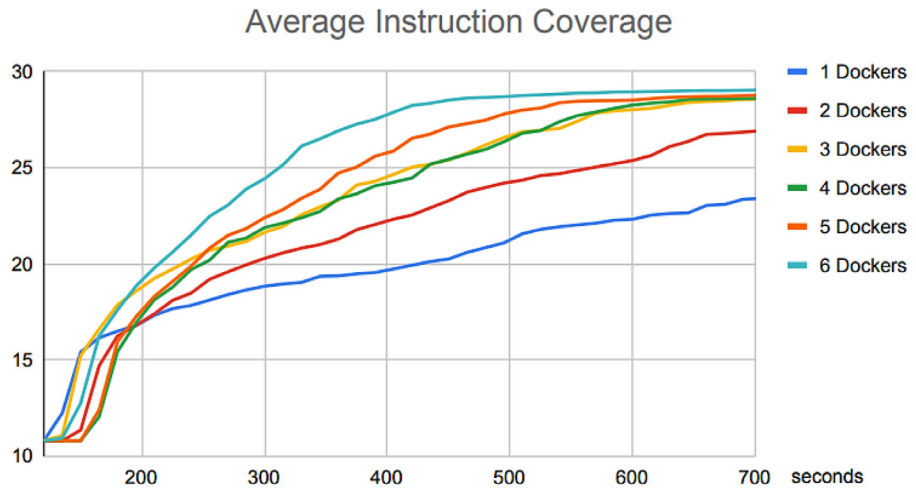

Fig. 8. Parabank average code coverage growth for each group of Dockers.

Table 9
Parabank significant difference at 360 s.
Wilcoxon: *p*-value (p), Cliff's δ: small(S), medium(M) and large (L) difference.

Cliff's δ Wilcoxon	Instruction coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L −0.507	L −0.895	L −0.895	L −0.99	L −1.0		M −0.436	L −0.898	L −0.861	L −0.979	L −0.999
2 Dockers	p 0, 004165		L −0.543	L −0.555	L −0.781	L −0.951	p 0, 007078		L −0.592	L −0.515	L −0.758	L −0.936
3 Dockers	p 0, 000011	p 0, 001754		N 0.03	M −0.413	L −0.858	p 0, 000013	p 0, 002707		S 0.162	M −0.441	L −0.847
4 Dockers	p 0, 000012	p 0, 000334	p 0, 926255		M −0.457	L −0.83	p 0, 000011	p 0, 000777	p 0, 462133		L −0.496	L −0.847
5 Dockers	p 0, 000010	p 0, 000027	p 0, 004165	p 0, 008283		L −0.668	p 0, 000010	p 0, 000032	p 0, 013704	p 0, 010542		L −0.652
6 Dockers	p 0, 000010	p 0, 000010	p 0, 000011	p 0, 000053	p 0, 000334		p 0, 000010	p 0, 000010	p 0, 000012	p 0, 000034	p 0, 000648	

Table 10
Parabank significant difference at 480 s.
Wilcoxon: *p*-value (p), Cliff's δ: small(S), medium(M) and large (L) difference.

Cliff's δ Wilcoxon	Instruction coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L −0.672	L −0.971	L −0.903	L −1.0	L −1.0		L −0.662	L −0.967	L −0.91	L −0.993	L −1.0
2 Dockers	p 0, 000047		L −0.541	L −0.531	L −0.812	L −1.0	p 0, 000108		L −0.6	L −0.554	L −0.828	L −1.0
3 Dockers	p 0, 000005	p 0, 002136		N 0.003	L −0.554	L −0.98	p 0, 000006	p 0, 003292		N 0.011	L −0.531	L −0.973
4 Dockers	p 0, 000008	p 0, 002982	p 0, 853135		L −0.508	L −0.95	p 0, 000006	p 0, 002482	p 0, 696972		L −0.486	L −0.946
5 Dockers	p 0, 000005	p 0, 000016	p 0, 000337	p 0, 003620		L −0.791	p 0, 000006	p 0, 000020	p 0, 000363	p 0, 005343		L −0.797
6 Dockers	p 0, 000005	p 0, 000005	p 0, 000005	p 0, 000005	p 0, 000041		p 0, 000006	p 0, 000006	p 0, 000006	p 0, 000006	p 0, 000026	

Table 11
Parabank significant difference at 600 s.
Wilcoxon: *p*-value (p), Cliff's δ: small(S), medium(M) and large (L) difference.

Cliff's δ Wilcoxon	Instruction coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L −0, 732	L −0, 988	L −0, 997	L −1, 0	L −1, 0		L −0, 735	L −0, 982	L −0, 993	L −1, 0	L −1, 0
2 Dockers	p 0, 00001		L −0, 913	L −0, 962	L −1, 0	L −1, 0	p 0, 000012		L −0, 912	L −0, 962	L −1, 0	L −1, 0
3 Dockers	p 0, 000005	p 0, 00003		S −0, 283	L −0, 496	L −0, 933	p 0, 000005	p 0, 000042		S −0, 275	L −0, 534	L −0, 941
4 Dockers	p 0, 000005	p 0, 00001	p 0, 034456		S −0, 271	L −0, 924	p 0, 000005	p 0, 000012	p 0, 018889		M −0, 33	L −0, 943
5 Dockers	p 0, 000005	p 0, 000005	p 0, 001463	p 0, 016825		L −0, 823	p 0, 000005	p 0, 000005	p 0, 001026	p 0, 010314		L −0, 836
6 Dockers	p 0, 000005	p 0, 000005	p 0, 000005	p 0, 000006	p 0, 000024		p 0, 000005	p 0, 000005	p 0, 000006	p 0, 000008	p 0, 000013	

Table 12
Parabank consumption average.

Dockers	1	2	3	4	5	6
CPU	16, 57%	23, 93%	29, 86%	36, 76%	41, 49%	47, 49%
MEM	23, 85%	29, 45%	34, 90%	40, 46%	45, 38%	50, 27%

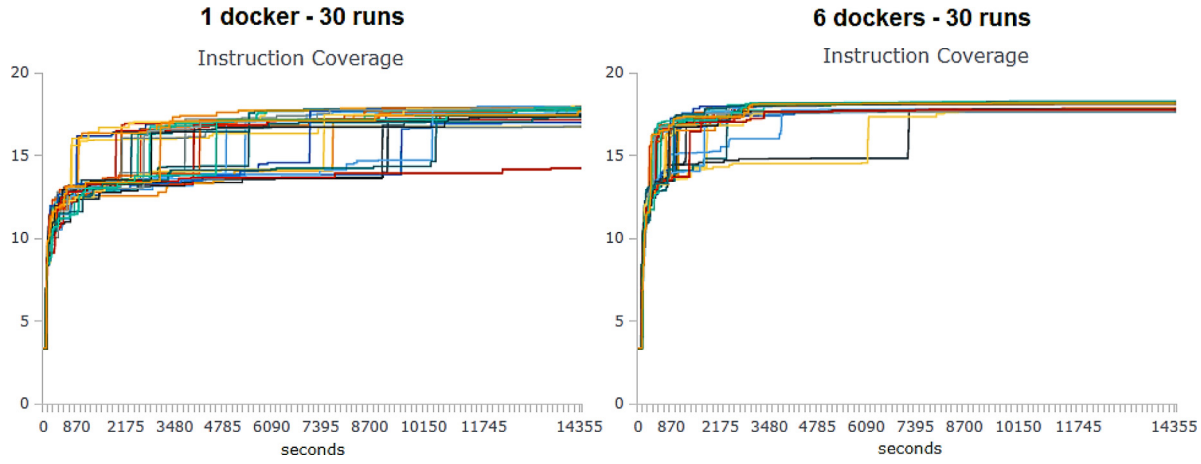

Fig. 9. Shopizer code coverage comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.

for one specific instance in the group of 5 dockers that reached 23.5% of coverage. We consider that a set of random actions led to purchasing or modifying a product in the store, increasing the code coverage.

For GUI coverage (see Fig. 10), the group of 6 Dockers needs approximately 1 h to discover the peak of abstract states. Then, we can observe a decrease in the unvisited abstract actions curve. Otherwise, the group of 1 Docker keeps discovering new states throughout the 4 h of execution. There is an execution time in which there is no noticeable decrease in the number of unvisited abstract actions.

With Shopizer, after 1 h of execution with a group of 6 Dockers, the number of unvisited actions slowly drops until 3 h of execution, when it seems to reach a limit. As with Parabank, we assume this is due to unreachable unvisited actions because of non-determinism in the model.

Fig. 11 represents the average code coverage of all Docker instances for Shopizer in the first hour of execution. We can see a growth difference between all groups of Dockers. However, the groups of 3, 4, 5, and 6 Dockers seem to differ only slightly.

We perform statistical analysis at 500, 1750 and 3000 s to obtain evidence about the significant difference in the distributed approach. From the moment the distributed instances have started exploring to the moment the groups with the most Dockers have reached the coverage limit. The Kruskal Wallis test indicates a significant difference between all distributed Docker groups in these three instants of time (see Table 13).

At 500 s (see Table 14), Wilcoxon indicates no significant difference for 2 & 3, 2 & 4, 3 & 4, 3 & 5, and 5 & 6 groups of Dockers, including 3 & 6 groups of Dockers only regarding branch coverage. Then a significant difference for the other groups of Dockers. And Cliff's delta points negligent difference for 3 & 4 and 5 & 6 groups of Dockers. Medium difference for 2 & 3, 2 & 4, 3 & 5, 3 & 6, 4 & 5, and 4 & 6 groups of Dockers, including 1 & 2 groups of Dockers only regarding branch coverage. Then a large difference for the other groups is shown.

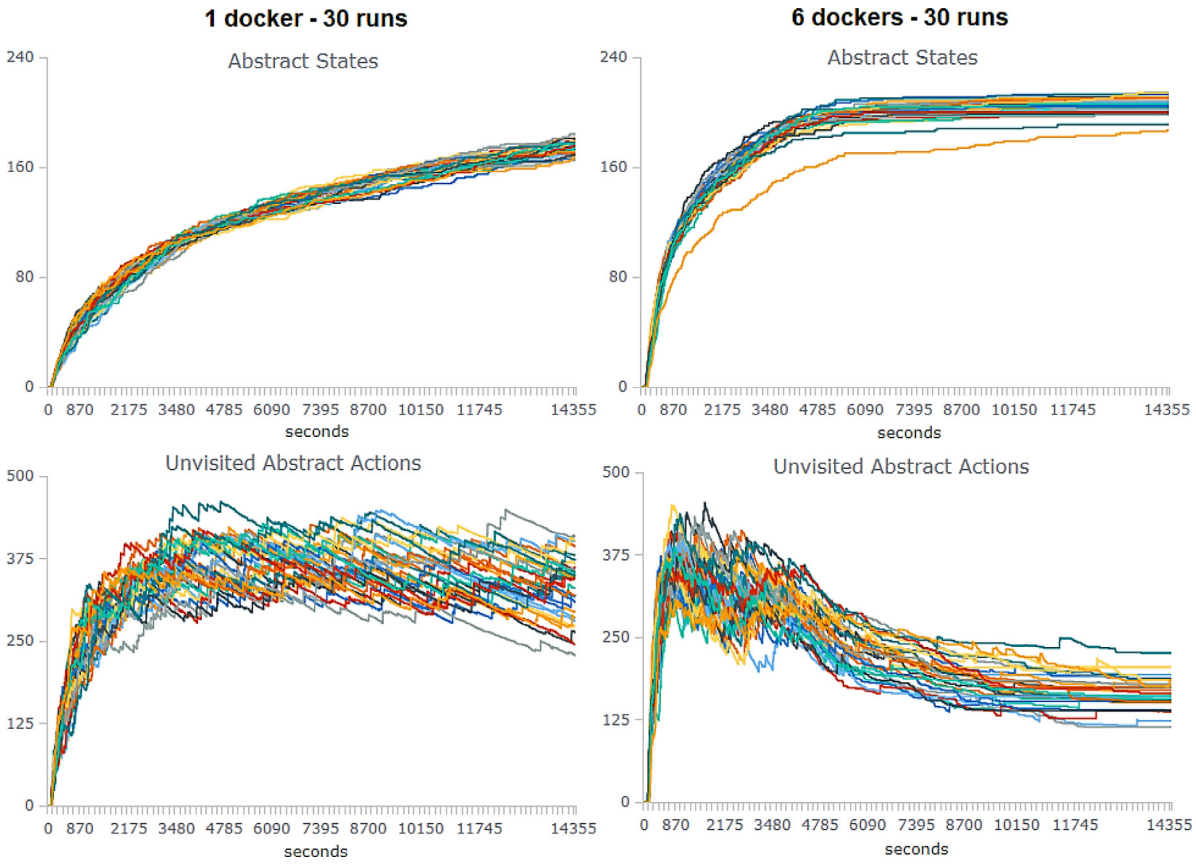

Fig. 10. Shopizer state model comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.

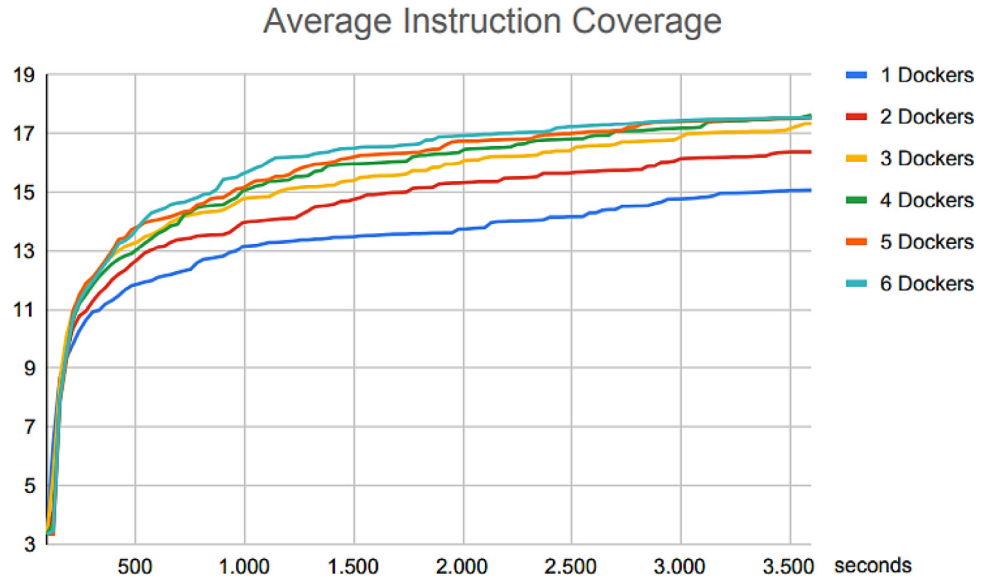

Fig. 11. Shopizer average code coverage growth for each group of Dockers.

Table 13
Shopizer Kruskal Wallis results.

Shopizer	Instruction coverage		Branch coverage	
	H-value	P-value	H-value	P-value
500 s	78.74	1.54e−15	74.06	1.46e−14
1750 s	82.53	2.48e−16	88.79	1.21e−17
3000 s	89.72	7.71e−18	105.74	3.25e−21

At 1750 s (see Table 15), Wilcoxon and Cliff's delta differ moderately from instruction and branch coverage. For instruction coverage, Wilcoxon specify no significant difference for 2 & 3, 3 & 4, 3 & 5, 4 & 5, 4 & 6, and 5 & 6 groups of Dockers and a significant difference for the other groups of Dockers. Whereas Cliff's delta points negligent difference for 4 & 5 and 5 & 6 groups of Dockers, small difference for 3 & 4 and 4 & 6, medium difference for 2 & 3 and 3 & 5 groups of Dockers, and large difference for the other groups.

Table 14
Shopizer significant difference at 500 s.
Wilcoxon: *p*-value (p), Cliff's δ: small(S), medium(M) and large (L) difference.

Cliff's δ Wilcoxon	Instruction coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L −0, 568	L −0, 796	L −0, 821	L −0, 962	L −0, 964		M −0, 464	L −0, 725	L −0, 76	L −0, 938	L −0, 91
2 Dockers	p 0, 003412		M −0, 395	M −0, 413	L −0, 703	L −0, 688	p 0, 017407		M −0, 434	M −0, 435	L −0, 708	L −0, 696
3 Dockers	p 0, 000042	p 0, 085892		N −0, 025	M −0, 37	M −0, 37	p 0, 000199	p 0, 086785		N 0, 012	M −0, 384	M −0, 346
4 Dockers	p 0, 000036	p 0, 096516	p 0, 869294		M −0, 395	M −0, 362	p 0, 000049	p 0, 080592	p 0, 638817		M −0, 457	M −0, 397
5 Dockers	p 0, 000019	p 0, 003217	p 0, 077814	p 0, 010939		N 0, 073	p 0, 000029	p 0, 003453	p 0, 080592	p 0, 005351		N 0, 087
6 Dockers	p 0, 000019	p 0, 001711	p 0, 047378	p 0, 015446	p 0, 869294		p 0, 000032	p 0, 001469	p 0, 080592	p 0, 016355	p 0, 643508	

Table 15
Shopizer significant difference at 1750 s.
Wilcoxon: *p*-value (p), Cliff's δ: small(S), medium(M) and large (L) difference.

Cliff's δ Wilcoxon	Instruction coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L −0, 72	L −0, 857	L −0, 902	L −0, 924	L −0, 944		L −0, 743	L −0, 862	L −0, 904	L −0, 928	L −0, 946
2 Dockers	p 0, 000112		M −0, 371	L −0, 558	L −0, 642	L −0, 717	p 0, 000089		M −0, 448	L −0, 582	L −0, 698	L −0, 773
3 Dockers	p 0, 000112	p 0, 070405		S −0, 252	M −0, 396	L −0, 481	p 0, 000114	p 0, 048391		S −0, 172	M −0, 426	L −0, 536
4 Dockers	p 0, 000082	p 0, 013455	p 0, 156805		N −0, 124	S −0, 18	p 0, 000082	p 0, 009743	p 0, 191964		S −0, 241	M −0, 364
5 Dockers	p 0, 000052	p 0, 000354	p 0, 052504	p 0, 614315		N −0, 08	p 0, 000058	p 0, 000172	p 0, 042723	p 0, 354629		N −0, 108
6 Dockers	p 0, 000052	p 0, 000711	p 0, 013455	p 0, 152613	p 0, 614315		p 0, 000058	p 0, 000316	p 0, 008778	p 0, 043083	p 0, 354629	

Table 16
Shopizer significant difference at 3000 s.
Wilcoxon: *p*-value (p), Cliff's δ: negligent (N), small(S), medium(M) and large (L) difference.

Cliff's δ Wilcoxon	Instruction coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L −0, 645	L −0, 851	L −0, 893	L −0, 93	L −0, 915		L −0, 674	L −0, 882	L −0, 913	L −0, 94	L −0, 928
2 Dockers	p 0, 001319		M −0, 446	L −0, 61	L −0, 732	L −0, 783	p 0, 001154		L −0, 592	L −0, 71	L −0, 832	L −0, 851
3 Dockers	p 0, 000028	p 0, 047193		S −0, 306	L-0, 522	L −0, 612	p 0, 000025	p 0, 01103		S −0, 316	L −0, 641	L −0, 76
4 Dockers	p 0, 000028	p 0, 003203	p 0, 115218		S −0, 232	S −0, 32	p 0, 000025	p 0, 000711	p 0, 075307		M −0, 335	L −0, 512
5 Dockers	p 0, 000028	p 0, 000028	p 0, 006422	p 0, 183593		N −0, 102	p 0, 000025	p 0, 000025	p 0, 001546	p 0, 049846		S −0, 221
6 Dockers	p 0, 000028	p 0, 000172	p 0, 007976	p 0, 047193	p 0, 673247		p 0, 000025	p 0, 000041	p 0, 002303	p 0, 004922	p 0, 265425	

Table 17
Shopizer consumption average.

Dockers	1	2	3	4	5	6
CPU	19.17%	26.08%	31.53%	36.29%	42.28%	46.14%
MEM	32.06%	38.88%	46.29%	52.9%	60.4%	67.3%

While, for branch coverage, Wilcoxon indicates no significant difference only for 3 & 4, 4 & 5, and 5 & 6 groups of Dockers and a significant difference for the other groups of Dockers. And Cliff's delta points negligent difference for 5 & 6 groups of Dockers, small difference for 3 & 4 and 4 & 5, medium difference for 2 & 3, 3 & 5, and 4 & 6 groups of Dockers, and large difference for the other groups.

Finally, *at 3000 s* (see Table 16), Wilcoxon and Cliff's delta continue differing from instruction and branch coverage. For instruction coverage Wilcoxon specify no significant difference for 3 & 4, 4 & 5, and 5 & 6 groups of Dockers. Whereas Cliff's delta points negligent difference for 5 & 6 groups of Dockers, small difference for 3 & 4, 4 & 5, and 4 & 6, medium difference for 2 & 3 groups of Dockers, and large difference for the other groups.

Then, for branch coverage, Wilcoxon indicates no significant difference only for 3 & 4 and 5 & 6 groups of Dockers. And Cliff's delta points small difference for 3 & 4 and 5 & 6, medium difference for 4 & 5 groups of Dockers, and large difference for the other groups.

Related to Shopizer resource consumption, Table 17 presents the CPU and memory consumption average of all groups of Dockers. CPU and memory consumption are increased around 4%–7% and 6%–8% respectively, for each new distributed Docker instance executed.

5.3. Answer to the research question

Statistical analysis of Parabank and Shopizer results allow us to answer *RQ1: Does a distributed architecture allow scriptless testing tools to speed up state model inference?* Kruskal Wallis, Wilcoxon and Cliff's delta tests demonstrate that using a distributed approach is significantly different throughout the state model inference process depending on the number of distributed instances used. Therefore, we can reject the H_0*: The distributed approach does not reduce the time required to infer a state model.*.

5.4. Threats to validity

This subsection mentions some threats that could affect the validity of our results (Wohlin et al., 2012; Ralph and Tempero, 2018; Feldt and Magazinius, 2010).

5.4.1. Face validity

We use code-level metrics to evaluate the efficiency of the distributed approach. Although these metrics have wide adoption in the GUI testing field for desktop and mobile applications (Pezzè et al., 2018; Su et al., 2017), they seem unusual in web domain GUI testing studies (Marchetto et al., 2011), apparently due to the lack of access to the server-side services (Coppola and Alégroth, 2022).

5.4.2. Content validity

Compared with other state-of-the-art research, this GUI web study focuses its empirical evaluation using code-level metrics. In future GUI studies, we will research the additional usage of model-level and GUI-level metrics (Coppola and Alégroth, 2022).

5.4.3. Internal validity

We detected that some distributed instances stopped receiving messages from the chromedriver renderer, which caused some Docker groups to decrease the number of working instances. These groups of Dockers were re-executed.

We decided to start extracting coverage metrics before building and deploying all involved software in the architecture of the experiment. Something that we consider to be closer to the industrial pipelines. Although this has disrupted the first seconds of coverage measurement, the subsequent coverage growth has shown that a distributed architecture improves according to the number of instances used despite having a different start time.

5.4.4. External validity

We use two web applications with a DMSS architecture to conduct the study, which accomplishes an essential part of today's web services, sharing resources (Choudhury, 2014). Even though we demonstrated the distributed approach helps improve the model inference efficiency, and we faced relevant GUI model inference challenges, to generalize our results properly, we will conduct future experiments with different types of desktop, web, and mobile systems.

5.4.5. Conclusion validity

Because there is a certain degree of randomness when choosing the actions to execute, we cannot assume normal distribution in the experiments (Arcuri and Briand, 2011). To deal with it, we have applied Kruskal Wallis, Wilcoxon, and Cliff's delta statistical non-parametric tests to the results obtained from the 30 runs of all Dockers groups.

5.4.6. Credibility

JaCoCo is used to obtain code coverage metrics. Since JaCoCo is an open source project widely adopted by the software research and industry development communities, we rely on its definitions of instruction and branch coverage and precise measurements (Ivanković et al., 2019; Silva et al., 2021).

6. Conclusions and future work

This paper presents a new technique that improves the speed of model inference in automated scriptless GUI testing, namely distributed state model inference. The DMSS architecture allows multiple instances to infer a shared state model. The proposed shared knowledge algorithm allows these distributed instances to coordinate and improve the inference speed by dividing the unvisited actions to be explored. Moreover, we have explained the abstraction challenges encountered, related to dynamism and non-determinism, and our solutions to reduce the impact on the distributed model inference.

To validate the distributed approach, we prepared an empirical experiment with two different open source web SUTs. The configuration of independent variables for each SUT is required for the model inference with a single instance or distributed approach. On the one hand, the results of our experiment indicate that a DMSS architecture improves the speed of the state model inference process. But on the other hand, it requires investing additional resources to prepare the distributed testing architecture and increase the hardware consumption. With the results presented in this paper, companies can decide if increasing resources to improve the inference speed is beneficial. To gather more evidence and offer more results, we expect to continue experimenting with a wider variety of SUTs, such as desktop and mobile, and other distributed architectures, such as DMIS.

In this study, we evaluate the efficiency of the distributed approach by measuring the code coverage of the SUT objects. In further experiments, we will research the usage of model-level, GUI-level metrics, and other test effectiveness metrics, such as the number of failures found, because a distributed approach may throw concurrent errors in the SUT that do not occur with a single instance.

To perform the distributed approach research, we handled the abstraction challenges regarding dynamism and non-determinism. Preparing the set of strategies to deal with the dynamism of each SUT requires effort to identify, implement and verify these strategies in the scriptless testing tool. Moreover, the presented non-deterministic shared knowledge ASM, still has the possibility of leaving non-deterministic parts of the model isolated. For these reasons, we plan further experimentation to find a suitable abstraction strategy for each SUT and evaluate the feasibility of automatically detecting dynamic widgets in the existing SUT states and restoring non-deterministic transitions at runtime. Additionally, we want to study the traceability of the models inferred with the distributed approach.

CRediT authorship contribution statement

Fernando Pastor Ricós: Conceptualization, Methodology, Software, Validation, Investigation, Writing – original draft, Review & editing. **Arend Slomp:** Conceptualization, Methodology, Software, Investigation. **Beatriz Marín:** Conceptualization, Methodology, Writing – original draft, Review & editing. **Pekka Aho:** Conceptualization, Methodology, Software, Writing – original draft, Review & editing. **Tanja E.J. Vos:** Conceptualization, Methodology, Writing – original draft, Review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Fernando Pastor Ricos, Beatriz Marin, Tanja Vos reports financial support was provided by Horizon 2020. Pekka Aho reports financial support was provided by Information Technology for European Advancement.

Data availability

DOI is included in the paper: https://doi.org/10.5281/zenodo.7607233. This DOI includes a replication package with instructions. https://nam11.safelinks.protection.outlook.com/?url=https%3A%2F%2Fdoi.org%2F10.5281%2Fzenodo.7607233&data=05%7C01%7Cs.ba%40elsevier.com%7C423c48cc07b542ea1fc008db0b4f83a1%7C9274ee3f94254109a27f9fb15c10675d%7C0%7C0%7C638116208362076629%7CUnknown%7CTWFpbGZsb3d8eyJWIjoiMC4wLjAwMDAiLCJQIjoiV2luMzIiLCJBTiI6Ik1haWwiLCJXVCI6Mn0%3D%7C3000%7C%7C%7C&sdata=PN3sXtmgvd9724qJJVzGWt0Z%2BpoL5a%2BrY1hxBbhS9yU%3D&reserved=0.

Acknowledgments

Thanks to Olivia Rodríguez-Valdés. This research has been funded by the following projects: H2020 EU project iv4XR (www.iv4xr-project.eu) grant nr. 856716, and ITEA3 project IVVES (www.ivves.eu) grant nr. 18022.

References

Aho, P., Alégroth, E., Oliveira, R.A., Vos, T.E., 2016. Evolution of automated regression testing of software systems through the graphical user interface. In: First International Conference on Advances in Computation, Communications and Services. International Academy, Research, and Industry Association (IARIA), pp. 16–21.

Aho, P., Buijs, G., Akin, A., Senturk, S., Pastor-Ricós, F., de Gouw, S., Vos, T.E., 2021. Applying scriptless test automation on web applications from the financial sector. Actas de Las XXV Jornadas de IngenieríA Del Software Y Bases de Datos (JISBD 2021) 1–4.

Aho, P., Suarez, M., Kanstrén, T., Memon, A.M., 2013. Industrial adoption of automatically extracted GUI models for testing. In: EESSMOD@ MoDELS. pp. 49–54.

Aho, P., Suarez, M., Kanstrén, T., Memon, A.M., 2014. Murphy tools: Utilizing extracted gui models for industrial software testing. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops. IEEE, pp. 343–348.

Aho, P., Suarez, M., Memon, A., Kanstrén, T., 2015. Making GUI testing practical: Bridging the gaps. In: 2015 12th International Conference on Information Technology-New Generations. IEEE, pp. 439–444.

Aho, P., Vos, T., 2018. Challenges in automated testing through graphical user interface. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, IEEE, pp. 118–121.

Arcuri, A., Briand, L., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 2011 33rd International Conference on Software Engineering. ICSE, IEEE, pp. 1–10.

Bauersfeld, S., Vos, T.E., Condori-Fernández, N., Bagnato, A., Brosse, E., 2014. Evaluating the TESTAR tool in an industrial case study. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–9.

Brach, P., Chrzaszcz, J., Jabłonowski, J., Światły, J., 2011. A distributed service oriented system for GUI map generation. In: Proceedings of the 12th International Conference on Computer Systems and Technologies. pp. 69–74.

Chahim, H., Duran, M., Vos, T.E., Aho, P., Condori Fernandez, N., 2020. Scriptless testing at the GUI level in an industrial setting. In: International Conference on Research Challenges in Information Science. Springer, pp. 267–284.

Choudhury, N., 2014. World wide web and its journey from web 1.0 to web 4.0. Int. J. Comput. Sci. Inf. Technol. 5 (6), 8096–8100.

Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P., 2011. Model checking and the state explosion problem. In: LASER Summer School on Software Engineering. Springer, pp. 1–30.

Coppola, R., Alégroth, E., 2022. A taxonomy of metrics for GUI-based testing research: A systematic literature review. Inf. Softw. Technol. 152, 107062. http://dx.doi.org/10.1016/j.infsof.2022.107062, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922001719.

CSTI, 2013, 2022. Shopizer v2.17 java e-commerce software. Last accessed: 7 Apr 2022 [Online]. Available: https://github.com/shopizer-ecommerce/shopizer.

Feldt, R., Magazinius, A., 2010. Validity threats in empirical software engineering research-an initial survey. In: Seke. pp. 374–379.

Furia, C.A., Feldt, R., Torkar, R., 2019. Bayesian data analysis in empirical software engineering research. IEEE Trans. Softw. Eng. 47 (9), 1786–1810.

de Gier, F., Kager, D., de Gouw, S., Vos, E.T., 2019. Offline oracles for accessibility evaluation with the TESTAR tool. In: 2019 13th International Conference on Research Challenges in Information Science. RCIS, IEEE, pp. 1–12.

Grilo, A.M., Paiva, A.C., Faria, J.P., 2010. Reverse engineering of GUI models for testing. In: 5th Iberian Conference on Information Systems and Technologies. IEEE, pp. 1–6.

Haoyin, L., 2017. Automatic android application GUI testing—A random walk approach. In: 2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET). IEEE, pp. 72–76.

Ivanković, M., Petrović, G., Just, R., Fraser, G., 2019. Code coverage at Google. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 955–963.

JaCoCo, 2006, 2022. Java Code Coverage. (Last accessed: 7 Apr 2022) [Online]. Available: https://www.jacoco.org.

Kropp, M., Meier, A., Anslow, C., Biddle, R., 2018. Satisfaction, practices, and influences in agile software development. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. pp. 112–121.

Luo, Q., Hariri, F., Eloussi, L., Marinov, D., 2014. An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 643–653.

Marchetto, A., Tiella, R., Tonella, P., Alshahwan, N., Harman, M., 2011. Crawlability metrics for automated web testing. Int. J. Softw. Tools for Technol. Transf. 13 (2), 131–149.

Mariani, L., Pezzè, M., Riganelli, O., Santoro, M., 2014. Automatic testing of GUI-based applications. Softw. Test. Verif. Reliab. 24 (5), 341–366.

Memon, A.M., 2007. An event-flow model of GUI-based applications for testing. Softw. Test. Verif. Reliab. 17 (3), 137–157.

Memon, A., Banerjee, I., Nguyen, B.N., Robbins, B., 2013. The first decade of gui ripping: Extensions, applications, and broader impacts. In: 2013 20th Working Conference on Reverse Engineering. WCRE, IEEE, pp. 11–20.

Mesbah, A., Van Deursen, A., Lenselink, S., 2012. Crawling ajax-based web applications through dynamic analysis of user interface state changes. ACM Trans. Web (TWEB) 6 (1), 1–30.

Mulders, A., Valdes, O.R., Ricós, F.P., Aho, P., Marín, B., Vos, T.E., 2022. State model inference through the GUI using run-time test generation. In: International Conference on Research Challenges in Information Science. Springer, pp. 546–563.

Nass, M., Alégroth, E., Feldt, R., 2021. Why many challenges with GUI test automation (will) remain. Inf. Softw. Technol. 138, 106625.

Parasoft, 2017, 2022. The ParaBank demo application from Parasoft. Last accessed: 7 Apr 2022 [Online]. Available: https://github.com/parasoft/parabank.

Pezzè, M., Rondena, P., Zuddas, D., 2018. Automatic GUI testing of desktop applications: An empirical assessment of the state of the art. In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops. ISSTA '18, Association for Computing Machinery, New York, NY, USA, pp. 54–62. http://dx.doi.org/10.1145/3236454.3236489.

Rafi, D.M., Moses, K.R.K., Petersen, K., Mäntylä, M.V., 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: 2012 7th International Workshop on Automation of Software Test. AST, IEEE, pp. 36–42.

Ralph, P., Tempero, E., 2018. Construct validity in software engineering research and software metrics. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. pp. 13–23.

Ricós, F.P., Aho, P., Vos, T., Boigues, I.T., Blasco, E.C., Martínez, H.M., 2020. Deploying TESTAR to enable remote testing in an industrial CI pipeline: a case-based evaluation. In: International Symposium on Leveraging Applications of Formal Methods. Springer, pp. 543–557.

Roest, D., Mesbah, A., Van Deursen, A., 2010. Regression testing ajax applications: Coping with dynamism. In: 2010 Third International Conference on Software Testing, Verification and Validation. IEEE, pp. 127–136.

Selenium, 2022. Docker images for the selenium grid server. (Last accessed: 7 Apr 2022) [Online]. Available: https://hub.docker.com/r/selenium/standalone-chrome.

Silva, A., Martinez, M., Danglot, B., Ginelli, D., Monperrus, M., 2021. FLACOCO: Fault localization for Java based on industry-grade coverage. arXiv preprint arXiv:2111.12513.

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., Su, Z., 2017. Guided, stochastic model-based GUI testing of android apps. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, pp. 245–256. http://dx.doi.org/10.1145/3106237.3106298.

Vos, T.E., Aho, P., Pastor Ricos, F., Rodriguez-Valdes, O., Mulders, A., 2021. Testar–scriptless testing through graphical user interface. Softw. Test. Verif. Reliab. 31 (3), e1771.

Vos, T.E., Marín, B., Escalona, M.J., Marchetto, A., 2012. A methodological framework for evaluating software testing techniques and tools. In: 2012 12th International Conference on Quality Software. IEEE, pp. 230–239.

Watada, J., Roy, A., Kadikar, R., Pham, H., Xu, B., 2019. Emerging trends, techniques and open issues of containerization: a review. IEEE Access 7, 152443–152472.

Wen, H.-L., Lin, C.-H., Hsieh, T.-H., Yang, C.-Z., 2015. Pats: A parallel gui testing framework for android applications. In: 2015 IEEE 39Th Annual Computer Software and Applications Conference. 2, IEEE, pp. 210–215.

Wetzlmaier, T., Ramler, R., Putschögl, W., 2016. A framework for monkey GUI testing. In: 2016 IEEE International Conference on Software Testing, Verification and Validation. ICST, IEEE, pp. 416–423.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer Science & Business Media.

X.Org, 2022. X.Org and X11 protocol, server, driver, library, and application development. (Last accessed: 7 Apr 2022) [Online]. Available: https://gitlab.freedesktop.org/xorg.

Fernando Pastor Ricós Scientific researcher and Ph.D. student working in the Valencian Institute of Artificial Intelligence (VRAIN) group at the Universitat Politècnica de València (UPV). He has been researching the usage of the automated GUI testing scriptless technique for desktop, web, mobile, and extended reality systems with the open-source TESTAR tool since 2017 within the European H2020 projects DECODER and iv4XR.

Arend Slomp Dynamics Customer Relationship Management (CRM) consultant and Azure architect with more than 5 years of experience in solution architecture, business process design, development, and automated testing. He finished his Master of Science degree at the Open University of the Netherlands in 2021

by researching the integration of a docker version of the TESTAR tool in a DevOps environment.

Beatriz Marín Ph.D. in Computer Science from Universitat Politècnica de Valencia Spain (2011), with more than 10 years of experience in teaching, academic management, and leading research projects. She has been project manager in companies, senior researcher, professor of undergraduate and graduate students, director of research projects, and during the last 3 years, coordinator of research and master grade at Universidad Diego Portales - Chile. Nowadays, she is working as senior researcher of the Valencian Institute of Artificial Inteligence (VRAIN), Spain. She has more than 50 scientific articles in the software engineering area, particularly in the areas of conceptual modeling, software testing, model-driven software development, empirical software engineering, and gamification. She participates actively in scientific committees of national and international conferences, and also in journals of recognized prestige.

Dr. Pekka Aho works as a research scientist at the Open University of the Netherlands. He has been researching automated testing through graphical user interface (GUI), state model inference and monkey testing since 2010. He joined the international development team of open source TESTAR tool in 2017 and received his doctoral degree from University of Oulu in Finland in 2019.

Tanja Vos Associate Professor,Ph.D. in Software Testing; Teacher and researcher in Software Testing at the Computation and Information Systems Department (DSIC) of the UPV and she is director of the Software Quality & Testing (SQ&T) group at the ProS center. She has more than 20 years of experience with teaching and researching formal methods and software testing. Besides teaching, Tanja has been involved in various research projects on software testing in an industrial setting. Moreover, she has successfully coordinated the EU-funded projects like (EvoTest 2006–2009, and FITTEST 2010–2013). Currently, she is coordinating the research, development and in exploiting of the TESTAR (www.testar.org) tool that came out of the FITTEST project. Related to education and innovation, she has been involved in various Erasmus initiatives about innovation and technology transfer (SUPORT 510432LLP-1-2010-1-IE-ERASMUS-ECUE, Innovative Trainer - 2012-1-GB2-LEO05-07860, SHIP 554187-EPP-1-2014-1-IE-EPPKA2-KA). Nowadays she is leading the European iNnovation AllianCe for TESting education (ENACTEST ERASMUS+ project 101055874).