Guess the State: Exploiting Determinism to Improve GUI Exploration Efficiency

Diego Clerissi, Giovanni Denaro, *Member, IEEE*, Marco Mobilio, and Leonardo Mariani, *Senior Member, IEEE*

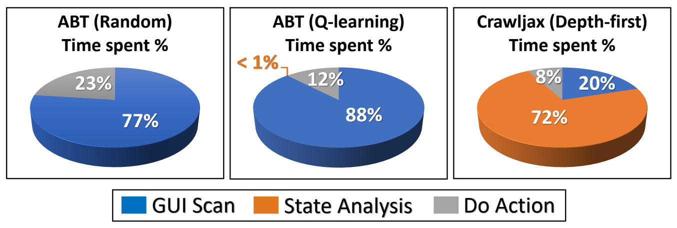
Abstract—Many automatic Web testing techniques generate test cases by analyzing the GUI of the Web applications under test, aiming to exercise sequences of actions that are similar to the ones that testers could manually execute. However, the efficiency of the test generation process is severely limited by the cost of analyzing the content of the GUI screens after executing each action.
In this paper, we introduce an inference component, SIBILLA, which accumulates knowledge about the behavior of the GUI after each action. SIBILLA enables the test generators to *reuse* the results computed for GUI screens that recur multiple times during the test generation process, thus improving the efficiency of Web testing techniques.
We experimented SIBILLA with Web testing techniques based on three different GUI exploration strategies (Random, Depth-first, and Q-learning) and nine target systems, observing reductions from 22% to 96% of the test generation time.

Index Terms—Web testing, System testing, State inference.

1 INTRODUCTION

WEB testing techniques generate test cases by interacting with the GUI of the Applications Under Test (AUT) [1], [2], [3], [4], [5], that is, performing actions on the graphical interface (e.g., clicking buttons or entering text in textfields) and checking the correctness of the responses (e.g., the expected text in a page). The simplest approaches, such as Monkey testing [6], randomly execute actions regardless of the content of the GUI, for instance producing click events at random screen coordinates. In this case, the test generator is particularly efficient in generating many events per second, but does not reason on the behavior of the application under test, and this limits its capability to generate complex sequences of interrelated events or capture failures that do not manifest as crashes [7], [8].

Other Web testing techniques (a) analyze the GUI screen of the AUT (*GUI scan*) after executing every action to extract the current AUT state; (b) compute state-specific properties (*state analysis*) relevant for their testing algorithms, e.g., whether the current state corresponds to a state already visited in the past; and (c) progress by selecting and executing an action that belongs to the current state (*do action*). The analysis of the GUI enables Web testing techniques that can reason on the generation of sophisticated sequences of interrelated interactions [9]. However, it also introduces a significant overhead in the test generation process. For instance, Figure 1 shows the distribution of the time spent for GUI scan, state analysis, and do action operations, for the Web testing techniques considered in the experiments that we report in this paper, when used to test the Dolibarr [10] Web application. Selecting and executing actions took only between 8% and 23% of the testing budget, while GUI scan and state analysis used most of the time (between 77% and 92%), with different relative contributions depending on the technique.



ABT (Random), left plot, the tool ABT [11] equipped with random strategy: It does not exploit any state analysis, simply executes random actions out of the ones that belong to each GUI screen;
ABT (Q-learning), mid plot, the tool ABT [11] equipped with Q-learning: It exploits state abstractions heuristically, to reward actions that are likely to explore sequences of distinguishable GUI screens;
Crawljax (Depth-first), right plot, the tool Crawljax [3]: It compares states by using precise distance metrics to identify recurring GUI states, to explore the possible action sequences in depth-first fashion.

Fig. 1: Relative time for GUI scan, state analysis, and do action operations, while testing the application Dolibarr [10].

This paper proposes a novel approach that increases the efficiency of Web test generators by drastically reducing the time allocated for GUI scan and state analysis operations. The key idea is to *identify and memoize the deterministic behaviors of the software*, in order to *infer recurring GUI states*, and thus *reuse the results of past GUI scan and state analysis computations* instead of executing them over and over again. In fact, Web test generators frequently repeat many interactions multiple times, while they incrementally unfold the execution space of the applications under test. For instance, every new test case starts from the same initial state and often re-executes early actions already done in previous test cases. This type of activity naturally leads to visit the same GUI states and perform the same actions multiple times, thus generating

• *D. Clerissi, G. Denaro, M. Mobilio and L. Mariani are with the Department of Informatics Systems and Communication, University of Milano - Bicocca, Milan, Italy.*
E-mail: {name}.{surname}@unimib.it

Manuscript received XXX YY, 20ZZ; revised XXX YY, 20ZZ.

opportunities for reuse.

Our approach, which we refer to as SIBILLA, can be incorporated in many Web test generators to monitor, memoize and reuse the results of the GUI scan and state analysis operations, and can thus deliver significant efficiency improvements for a wide range of Web testing techniques. SIBILLA maintains a knowledge base that represents the relation between the actions and the GUI states executed throughout the test generation process, and exploits the knowledge base to infer when the test generator is executing an action that deterministically leads to a GUI state that is already maintained in the knowledge base. For instance, SIBILLA can infer that clicking on a menu entry always leads to the same GUI state (e.g., a GUI page that lists the possible operations related to the given menu entry). In these cases, SIBILLA returns the GUI state and the information already computed correspondingly, thus superseding the need of both running the GUI scan operation and analyzing the resulting state.

Moreover, SIBILLA is designed to tolerate weakly non-deterministic behaviors, that is, behaviors that may lead to multiple - but finitely enumerable - GUI states. For instance, SIBILLA could infer that the output of a login request could be either the user homepage or a login-rejected page, and suitably memoize both these behaviors in its knowledge base. The non-determinism tolerance capacity, that is, the maximum number of alternative results that SIBILLA may handle for non-deterministic behaviors, is a configuration parameter. Upon detecting behaviors that exceed its non-determinism tolerance capacity, SIBILLA falls back to executing the usual GUI scan and state analysis operations. SIBILLA can also be configured with abstraction functions to optimize the space-efficiency of the knowledge base at the cost of some loss of precision of its inferences.

We experimented with SIBILLA equipped with an initial set of alternative abstraction functions, integrating it with three Web test generation techniques: Crawljax [3], which explores the action sequences of the application under test in depth-first fashion,[1] ABT [11], which derives test cases driven with the Q-learning-based heuristic of rewarding actions that likely explore sequences of distinguishable GUI screens, and Random, which executes actions chosen at random among the ones that belong to each GUI screen.

We tested nine Web applications with these three techniques, comparing the original versions of their algorithms with the corresponding versions integrated with SIBILLA, in turn equipped with several possible abstraction functions. The results show that SIBILLA is able to improve the efficiency of the considered Web testing techniques (often halving their execution time, and sometime reaching up to a 90% reduction), although it can sometimes introduce a limited degree of imprecision.

The paper is organized as follows. Section 2 introduces SIBILLA and its core algorithms. Section 3 presents the empirical evaluation of SIBILLA and discusses the obtained results. Sections 4 and 5 report related work and our conclusions, respectively.

2 THE SIBILLA APPROACH

2.1 Definitions

We first introduce some definitions that are useful to illustrate SIBILLA.

Definition 1. GUI state. A GUI state $s \in S$ is a set of widgets (i.e., graphical objects) $s \equiv \{w_1, w_2, ...\}$ that includes all the widgets $w_1, w_2, \dots$ visualized in a GUI screen. Each widget w is defined by its set of properties $w = \{p_i\}$, being each property a key-value pair $p_i = \langle k_i, v_i \rangle$, e.g., ⟨clickable, true⟩

Definition 2. Scan. SCAN is an operation SCAN : $SCR \rightarrow S$, that analyzes the GUI screen currently visualized ($scr \in SCR$) and returns the GUI state ($s \in S$) representing the set of the widgets that are part of the screen.

For instance, a GUI screen of a Web application can be a page with widgets that correspond to the Web forms, the text fields, and the buttons within the page. Each widget is characterized by several properties. For instance, button-widgets are characterized by properties such as ⟨clickable, true⟩ and ⟨value, 'submit'⟩. The SCAN operation returns the widgets in the current GUI state in the form of programmatic objects that the test generator can suitably inspect, store and exploit.

In particular, to progress in exploring the possible GUI states, the Web testing techniques select and execute actions by interacting with the widgets. To this end, since the chosen actions shall be then executed on the actual screen, the testing tools must be able to dereference those programmatic widgets with respect to the current screen. The technical means to dereference a widget is referred to as a *locator* [17].

Definition 3. Locator. A locator l is a programmatic object that uniquely identifies a widget in a given GUI state, and that can be dereferenced with respect to a current GUI screen to execute the actions of the widget.

Locators can take different forms, such as XPath descriptors and sets of properties that uniquely identify the specific widget [17].

Definition 4. Action. We define an action $a \in A$ as a triple $a = \langle l_w, type, P \rangle$, where l_w is the locator of the widget w targeted by the action, $type$ is the type of the action (e.g., click), and P is a (possibly empty) list of parameter values.

Some techniques exploit also compound actions consisting of sequences of interactions that must be executed without interruption (e.g., enter data in every input field present in a form and then submit the form).

2.2 Overview of SIBILLA

The SIBILLA approach that we propose in this paper draws on the observation that *GUI-based test generation techniques*

1. Crawljax is more of a Web crawling infrastructure, rather than a proper Web test generator, and some researchers reported issues in turning the action sequences explored by Web crawling into reproducible test suits [12], [13], [14]. Nonetheless, several researchers have used Crawljax as the basis for Web testing techniques [15], [16]. In this paper we are interested in how our approach can help optimize the exploration strategies that Web test generators may exploit for supporting their test generation strategies, and this makes Crawljax a very good candidate for experimenting the effectiveness of our approach. In this respect, and just for the sake of simplifying the presentation, we slightly abuse the terminology *Web test generator* when referring to Crawljax in the context of this paper.

tend to visit the same GUI states many times over and over again while exploring the GUI of the target application. Thus, when landing in a GUI state that has been already visited, a test generation technique could benefit of *reusing the result of the operations executed in the past* for that same state. Since executing the SCAN and state analysis operations are often computationally expensive, achieving this type of time-saving many times promises significant efficiency gains for the test generation process.

To this end, SIBILLA populates and maintains a knowledge base of the GUI states that a test generation technique visits during a testing session, and exploits the knowledge base at each new GUI interaction to try to *infer the reached state, superseding the need of both executing the SCAN operation and performing the associated computations*. In the following, we introduce how SIBILLA works incrementally, starting from a naive definition, which captures the essential idea of SIBILLA of leveraging deterministically recurring interactions, up to its final definition, which aims to address the spectrum of challenges to foster efficient Web testing techniques.

SIBILLA naive: In its most naive form, the SIBILLA knowledge base is a map that associates a GUI interaction with the results of both executing SCAN and performing state analysis at the reached GUI screen. More rigorously, the naive knowledge base has the form

$$\text{SIBILLA}_{naive} : S \times A \rightarrow S \times D$$

where S denotes GUI states, A denotes possible actions, and D generically denotes the state data that the test generator can compute with state analysis. $\text{SIBILLA}_{naive}(s,a) = \langle s', d' \rangle$ means that $s' \in S$ is the GUI state reached by executing the action a from state s, and $d' \in D$ is the result of state analysis for s'. Note that the actual content of the data d' depend on the specific test generator being considered. For instance, Crawljax, one of the tools that we used in our experiments, computes the edit-distance [18] between states to identify already visited states (i.e., the ones with zero-distance from some previously visited state). This check becomes increasingly expensive while the number of explored GUI states increases over time. In this case, the data d' can store the result of the checks already done to avoid recomputing them when a given state is reached again.

The knowledge base is populated incrementally while actions are executed. In particular, when a testing technique executes an action a from a state s, it can refer to SIBILLA_{naive} to check whether that interaction is already part of the knowledge base. If this is the case, SIBILLA_{naive} returns the corresponding GUI state along with the associated data from the knowledge base without executing neither SCAN nor state analysis. Otherwise, if the interaction is not yet in the knowledge base, SIBILLA_{naive} executes SCAN, analyzes the resulting state, and adds the results in the knowledge base for future reuse opportunities.

For instance, let us consider a Web application under test in a state in which the application visualizes a login screen as the one sketched in the left side of Figure 2. Let $\hat{s}$ be the GUI state that the SCAN operation returns for this screen, that is, $\hat{s}$ consists of a page with title "Login page", two label-widgets named "username" and "password", respectively, two textfield-widgets filled with the strings "alice" and

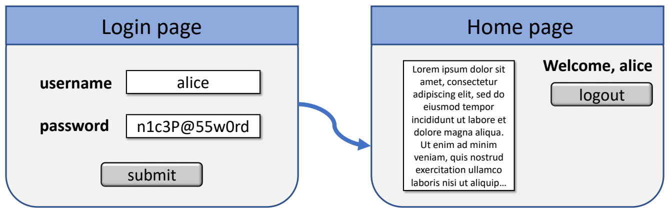

Login page
username
alice
password
n1c3P@55w0rd
submit
Home page
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip...
Welcome, alice
logout

Fig. 2: A sample GUI interaction.

"n1c3P@55w0rd", respectively, and a button-widget to submit the inquiry. SIBILLA_{naive} can observe that when the submit-action is executed from that state, the application lands in a new screen like the one in the right part of the figure, which SCAN returns like the GUI state $\tilde{s}$, and the analysis of the state generates the data $\tilde{d}$. Thus, SIBILLA_{naive} stores the entry $\langle \hat{s}, \hat{a} \rangle \rightarrow \langle \tilde{s}, \tilde{d} \rangle$ in its knowledge base: if a testing technique executes the same interaction once again, that is, if it submits another login inquiry with the same name and password as in the figure, SIBILLA_{naive} can return the state $\tilde{s}$ along with the associated data $\tilde{d}$ without the need of scanning the GUI screen and analyzing the resulting state. Similarly, SIBILLA_{naive} could store the attempts with wrong username-password pairs that keep the GUI state in the login page, and reuse those results.

SIBILLA non-deterministic: SIBILLA_{naive} assumes that the GUI interactions are strictly deterministic. However, the determinism of the GUI interactions cannot be taken for granted because the GUI state is generally a partial representation of the execution state of the application under test. For example, the interaction that we exemplified with reference to Figure 2 holds only until the user "alice" does not change her password. Conversely, after the password gets changed, the same interaction that we denoted above as $\langle \hat{s}, \hat{a} \rangle$ keeps the application in the login page, invalidating the entry $\langle \hat{s}, \hat{a} \rangle \rightarrow \langle \tilde{s}, \tilde{d} \rangle$ in the knowledge base.

SIBILLA acknowledges that some GUI interactions may yield non-deterministic results, and includes support for capturing the non-deterministic interactions to some degree. On one hand, it allows for associating an interaction with multiple distinct landing states, up to a bounded number, thus capturing non-deterministic interactions. On the other hand, it discriminates only those *non-deterministic behaviors with observable impact*, thus absorbing the differences that do not produce observable impacts on the GUI exploration mechanism of the testing technique.

In detail, the knowledge base of SIBILLA is designed to associate multiple states with each interaction, up to some (configurable) maximum number of states, thus allowing for tolerating some non-deterministic behaviors. Beyond the tolerance capacity, that is, when an interaction results in more distinct states than the maximum number, SIBILLA safely falls back to the nominal case of executing SCAN and state analysis. To this end, the SIBILLA knowledge base has the form

$$\text{SIBILLA}^{n}_{nondet} : S \times A \rightarrow \langle S \times D \rangle^{i \leq n} \cup n.d.$$

where, incrementally on SIBILLA_{naive}:

- n is a positive natural number that SIBILLA_{nondet} takes as configuration parameter,

- $\langle \dots \rangle^{i \le n}$ denotes a tuple of max n entries,
- a mapping like $\text{SIBILLA}^{n}_{nondet}(s,a) = \langle \langle s_1, d_1 \rangle, \langle s_2, d_2 \rangle, ..., \langle s_i, d_i \rangle \rangle$ means that, based on the interactions observed so far, executing the interaction $\langle s, a \rangle$ led to a GUI state out of the set $\{s_1, s_2, ..., s_i\}$, with $d_1, d_2, ..., d_i$ being the corresponding state analysis data, respectively, and
- a mapping like $\text{SIBILLA}^{n}_{nondet}(s,a) = n.d.$ means that the interaction is marked as non-deterministic beyond the tolerance capacity of $\text{SIBILLA}^{n}_{nondet}$, since the interactions observed so far revealed more than n distinct states.

When the testing technique executes a GUI interaction, $\text{SIBILLA}^{n}_{nondet}$ checks whether that interaction is part of the knowledge base. If the interaction is being executed for the first time, i.e., it is not in the knowledge base yet, $\text{SIBILLA}^{n}_{nondet}$ executes SCAN, analyzes the resulting state, and adds the results in the knowledge base. Otherwise, if the interaction is associated with a list of states (and is not $n.d.$), then $\text{SIBILLA}^{n}_{nondet}$ returns the first entry from the corresponding tuple. $\text{SIBILLA}^{n}_{nondet}$ optimistically assumes that the returned state conforms with the actual GUI state, but double-checks this assumption when the testing technique dereferences the widgets that belong to the state. Failing to dereference a widget reveals a non-conforming state with observable impact. In this case, $\text{SIBILLA}^{n}_{nondet}$ iterates and returns the next entry in the associated list, letting the tool re-try with the new state. $\text{SIBILLA}^{n}_{nondet}$ iterates this behavior until it returns a conforming state, or there are no other states in the list. When no conforming state is found, $\text{SIBILLA}^{n}_{nondet}$ executes the SCAN operation, analyzes the state, and either adds the results in the list, if it currently contains less then n states, or replaces the current list with the special value $n.d.$, to record that it observed more than n alternative states for the considered interaction and that both SCAN and state analysis must be always executed, if the current interaction is encountered again.

SIBILLA with abstractions: $\text{SIBILLA}^{n}_{nondet}$ does not control for the potentially unjustified proliferation of entries in the knowledge base. In fact, although an action may deterministically lead to a given GUI state, any small difference in distinct instances of that action (e.g., a small difference in the text entered in an input widget) can prevent reuse. Similarly, any small difference in distinct instances of the GUI states where the actions are executed can prevent the reuse opportunities. To address this challenge, SIBILLA relies on abstraction functions that can effectively collapse multiple concrete interactions with equivalent behavior into single abstract one. Abstraction functions can be exploited both to abstract from irrelevant details about actions and states, and to define approximated state matching heuristics. With the abstraction functions, the SIBILLA knowledge base further refines as:

$$\text{SIBILLA}^{n,absS,absA}_{abstract} : S_{abs} \times A_{abs} \rightarrow \langle S \times D \rangle^{i \le n} \cup n.d.$$

where, incrementally on $\text{SIBILLA}^{n}_{nondet}$:

- $absS : S \rightarrow S_{abs}$ is an abstraction function on the GUI states,
- $absA : A \rightarrow A_{abs}$ is an abstraction function on the GUI actions,
- a mapping like $\text{SIBILLA}^{n,absS,absA}_{abstract}(s_{abs}, a_{abs}) = \langle \langle s_1, d_1 \rangle, \langle s_2, d_2 \rangle, ..., \langle s_i, d_i \rangle \rangle$ means that, based on the interactions observed so far, executing any interaction $\langle s, a \rangle$ that corresponds to the abstract interaction $\langle s_{abs} = absS(s), a_{abs} = absA(a) \rangle$ led to a GUI state in the set $\{s_1, s_2, ..., s_i\}$ and to the corresponding state analysis data.

Working with $\text{SIBILLA}_{abstract}$, the SIBILLA knowledge base stores interactions in abstract format. Thus multiple concrete interactions may correspond to the same abstract interaction, and SIBILLA records the GUI states associated with the same abstract interaction as non-deterministic alternatives for that interaction.

For instance, referring again to the login screen of Figure 2, let us exemplify two abstraction functions, that is, the abstraction function $absS_{page}$ that abstracts all GUI states that correspond to the same Web page as the same abstract GUI state, and the abstraction function $absA_{locator}$ that abstracts all actions with the same locators as the same abstract action. With these abstraction functions, $\text{SIBILLA}^{n,absS,absA}_{abstract}$ records all submissions of login inquiries as the same abstract interaction, independently from the actual username and password values in the current screen. Yet, $\text{SIBILLA}^{n,absS,absA}_{abstract}$ stores all resulting GUI states (i.e., the screens that correspond to either a successful or a failed login operation, respectively) in the list with the abstract login interaction. This configuration of the knowledge base is particularly convenient, in fact it allows $\text{SIBILLA}^{n,absS,absA}_{abstract}$ to infer the concrete state reached after login in one or maximum two attempts, without the need of running the SCAN operation.

Choosing appropriate abstraction functions can have a great impact on the performances of SIBILLA, with the optimal choice being context dependent in most cases, as also noted in other papers that studied the impact of state abstractions [3], [12], [19]. For this reason, our current prototype of SIBILLA is equipped with multiple abstraction functions, which can be selected as configuration parameters, and is designed to be easily extensible to work with further abstraction functions that could be defined in the future for different use cases.

Note that abstractions on states and actions act synergistically and simultaneously, since an interaction corresponds to executing an (abstract) action when being in a given (abstract) state. In fact, significantly abstracting GUI states but not GUI actions might generate models with a proliferation of concrete actions that reach the same very abstract states, failing to distinguish the effect of the actions represented in the model. Viceversa, significantly abstracting GUI actions but not GUI states might lead to models with few actions non-deterministically reaching many diverse states, resulting in models where the state resulting from an action cannot be predicted anymore. Our current prototype of SIBILLA allows engineers to plug their own abstraction functions, to specifically customize their test generation technique, but we recommend to choose pairs of abstraction functions working at similar abstraction levels.

In the rest of the paper, we use the short name SIBILLA to refer to the full version $\text{SIBILLA}^{n,absS,absA}_{abstract}$ of the approach.

2.3 Reuse Opportunities with State Data

As discussed above, SIBILLA lets the test generation techniques benefit of reuse opportunities relatively to the state data that the test generator shall compute multiple times for the recurring GUI states. When the testing tool executes a GUI interaction that matches with an abstract interaction recorded in the knowledge base, SIBILLA returns a candidate GUI state, along with the associated state data.

For instance, in our experience with integrating SIBILLA in the test generation tools ABT and Crawljax, we successfully exploited the data associated with states to save the cost of the state-dependent computations that are performed by these tools. Here we elaborate on the case of Crawljax to give a concrete example.

Crawljax explores the action sequences of a Web application in depth-first order. At each new GUI state, it checks whether the current state corresponds to some previously traversed GUI state, aiming to discriminate the actions of the new state that loop to states already executed at previous traversals, and thus limit the next exploration steps to non-cyclic actions. To this end, Crawljax can be configured for using an adapted version of the traditional edit-distance algorithm [18], to identify the matching states as states with very high similarity, i.e., edit-distance below a given (low) threshold. Unfortunately, this check becomes increasingly expensive while the number of already explored GUI states increases over time.

We thus adapted Crawljax to store in the SIBILLA knowledge base the edit-distance values computed at each GUI state. Thus, when SIBILLA returns a GUI state from the knowledge base, Crawljax can reuse the already computed similarity results, and shall compute the edit-distance values only with respect to the GUI states visited since the last time it met that GUI state.

In general, the knowledge base can be used to store information that is either always the same for each GUI state, or varies monotonically at every new visit of a GUI state. This latter (monotonic state data) is in fact the case of the edit-distance values of Crawljax.

2.4 The SIBILLA Algorithm

In this section, we give the operational definition of the steps of SIBILLA that we have informally introduced in the previous section, and discuss how the GUI exploration algorithm of a test generation technique can be adapted to integrate with SIBILLA.

Algorithm 1 represents a (generic) GUI exploration algorithm that exploits SIBILLA in pseudocode: the non-shadowed pseudocode renders the nominal steps of the algorithm (i.e., the steps through which the algorithm would proceed even without SIBILLA), the shadowed steps depend on SIBILLA, and those in box-frames are part of the nominal algorithm but are not needed with SIBILLA. In particular, the functions SIBILLAGET and SIBILLAADDDATA (called at lines 11, 16, and 20) represent the execution of SIBILLA. Below, we first describe the integration logic by discussing Algorithm 1, and then formalize the definition of SIBILLA with reference to Algorithm 2.

In Algorithm 1, the nominal algorithm is intentionally generic in specifying how the considered technique accomplishes its test generation purposes. Essentially, the nominal algorithm simply describes a traversal of GUI states that starts from an initial GUI state and an initial action (lines 2–3), and then executes actions iteratively (line 6) until meeting some termination condition (e.g., reaching the maximum number of actions to execute, line 25). After each action, the algorithm relies on the SCAN operation to retrieve the GUI state that corresponds to the current screen (line 7), selects a new action to execute thereon (line 12), possibly extracts data from the GUI state (line 15), executes some tool-specific test generation behavior (line 23), and sets the current GUI state and the new action as the ones to consider at the next iteration (line 24).

Algorithm 1 GUI exploration with SIBILLA.

```
1:  function EXPLOREGUI
2:      $s \leftarrow$ Initial GUI state
3:      $a \leftarrow$ Initial action, selected in state $s$
4:      $KB \leftarrow \emptyset$
5:      do
6:          Execute action $a$ in state $s$
7:          $\boxed{s' \leftarrow \text{SCAN}}$
8:          $i \leftarrow 0$
9:          do
10:             $i \leftarrow i + 1$
11:             $s', d' \leftarrow \text{SIBILLAGET}(KB, i, s, a)$
12:             $a' \leftarrow$ Next action, selected in state $s'$
13:         while $a'$ is not valid in the current GUI screen
14:         if $d' = nil$ then
15:             $d' \leftarrow$ Compute data in $s'$
16:             $\text{SIBILLAADDDATA}(KB, s, a, s', d')$
17:         else
18:             if Refinement needed for $d'$ then
19:                 $d' \leftarrow$ Refine $d'$ based on $s'$
20:                 $\text{SIBILLAADDDATA}(KB, s, a, s', d')$
21:             end if
22:         end if
23:         Execute tool behavior against $s'$, $d'$
24:         $s \leftarrow s'$, $a \leftarrow a'$
25:     while Exploration not completed
26: end function
```

The algorithm integrated with SIBILLA exploits the SIBILLA knowledge base to try to superseed the need of performing the SCAN and state-analysis operations. Algorithm 1 renders this integration by defining an initially empty knowledge base (line 4), and replacing the direct invocation of SCAN in line 7 with the loop at lines 8–25 that iteratively queries the knowledge base for the candidate GUI states associated with the last executed action. The call to function SIBILLAGET (line 11) represents the query to the knowledge base to retrieve the i-th candidate GUI state associated with the pair $\langle s, a \rangle$, where s is the previous GUI state and a is the last executed action. The algorithm then validates the candidate state by checking whether the next action, selected out of the candidate state (line 12), can be correctly dereferenced against the current GUI screen (line 13). Otherwise, it iterates with the next candidate state. If the current knowledge base does not suffice to infer any valid candidate state for (s, a), SIBILLAGET falls back to executing SCAN and returns the resulting state (s', nil) to enrich the knowledge base. For instance, if the knowledge base includes 3 candidate states for a pair (s, a) and all of them are not valid for the current GUI screen, when invoking SIBILLAGET with $i = 3$, the functions falls back to using the SCAN operation to return a valid state.

When integrated with SIBILLA, the exploration algorithm can exploit the knowledge base also to retrieve state data that were already returned with the current GUI state (result d' at line 11). When this is the case, SIBILLA limits the computation of the data related to the current GUI state (nominal algorithm, line 15) only to the first time that a GUI state was added to the knowledge data (in this case the result d' at line 11 would be nil, thus enabling the check at line 14). The steps at line 18 and line 19 serve for those Web testing algorithms that must monotonically refine the previously computed state data when re-traversing a GUI state, as we exemplified in the previous section in the case of Crawljax. In these cases, the exploration algorithm replaces line 15 of the nominal algorithm with the less expensive step that refines the data retrieved from the knowledge base (line 19). Upon either computing or refining the state data, the exploration algorithm calls function SIBILLAADDDATA to update the content of the knowledge base to memoize the new or refined data (line 16 and line 20, respectively).

Algorithm 2 describes the functions SIBILLAGET, SIBILLAADDSTATE amd SIBILLAADDDATA that formalize the definition of SIBILLA in operational style.

SIBILLAGET takes as input the current knowledge base (parameter KB), the information on the last executed action, which includes both the action itself (parameter a) and the GUI state in which it has been executed (parameter s), and the index (parameter i) to access a specific candidate GUI state out of the list of candidate states that the knowledge base is able to match with the executed action. It returns the candidate GUI state retrieved from the knowledge base (output s'), along with the associated data available in the knowledge base (output d').

SIBILLAGET depends on the SCAN operation (line 6) and the abstraction functions for states and actions (lines 7 and 8). SIBILLA may fall back to executing SCAN when the knowledge base does not suffice to infer valid candidate states (line 27). SIBILLA uses the abstraction functions to mitigate the proliferation of entries and quickly accumulate knowledge about the behavior of the application. In fact, SIBILLAGET works with abstract states (s_{abs} at line 10) and actions (a_{abs} at line 11).

SIBILLAGET retrieves the candidate state from the knowledge base provided that the pair $\langle s_{abs}, a_{abs}\rangle$ refers to a knowledge base entry that both (a) was not previously marked as *non-deterministic* (line 12) and (b) matches with at least i candidate states, meaning that it indeed has an i-th candidate (lines 13–14). If this is the case, SIBILLAGET returns the i-th GUI state associated with $\langle s_{abs}, a_{abs}\rangle$ (lines 15–16).

Otherwise, SIBILLAGET falls back to executing SCAN, either in case of non-deterministic entries (i.e., when the check at line 12 fails, and it then calls SCAN at line 22) or in absence of matching candidates (i.e., when the check at line 14 fails, and it then hands the control to the subroutine SIBILLAADDSTATE that calls SCAN at line 27). In both these cases, being it the first time that the considered GUI state occurs, SIBILLAGET returns the newly scanned GUI state (line 23 and line 19, respectively), but does not return any associated data (it returns the special value nil), leaving for the testing tool the task of computing the data to associate with the new state.

Algorithm 2 Inferring GUI states with SIBILLA.

```
Input:
 1: $KB$: the SIBILLA knowledge base
 2: $s$: the GUI state before executing the last action
 3: $a$: the last executed action
 4: $i$: the index to access the list of candidate GUI states
Output:
 5: $s'$, $d'$: the inferred GUI state $s'$ with associated data $d'$
Dependencies:
 6: function SCAN                                          ▷ Extern
 7: function ABSS($s$)                                     ▷ Extern
 8: function ABSA($a$)                                     ▷ Extern

 9: function SIBILLAGET($KB, i, s, a$)
10:     $s_{abs} \leftarrow$ ABSS($s$)
11:     $a_{abs} \leftarrow$ ABSA($a$)
12:     if $KB(s_{abs}, a_{abs}) \neq n.d.$ then
13:         $Cndts \leftarrow KB(s_{abs}, a_{abs})$
14:         if $i \leq |Cndts|$ then
15:             $s', d' \leftarrow Cndts[i]$
16:             return $s', d'$
17:         else
18:             $s' \leftarrow$ SIBILLAADDSTATE($KB, s_{abs}, a_{abs}, Cndts$)
19:             return $s', nil$
20:         end if
21:     else
22:         $s' \leftarrow$ SCAN
23:         return $s', nil$
24:     end if
25: end function

26: function SIBILLAADDSTATE($KB, s_{abs}, a_{abs}, Cndts$)
27:     $s' \leftarrow$ SCAN
28:     $KB \leftarrow KB - \langle s_{abs}, a_{abs}, Cndts\rangle$
29:     if $|Cndts| = MAXSIZE$ then
30:         $KB \leftarrow KB \cup \langle s_{abs}, a_{abs}, n.d.\rangle$
31:     else
32:         $Cndts' \leftarrow append(Cndts, \langle s', nil\rangle)$
33:         $KB \leftarrow KB \cup \langle s_{abs}, a_{abs}, Cndts'\rangle$
34:     end if
35:     return $s'$
36: end function

37: function SIBILLAADDDATA($KB, s, a, s', d'$)
38:     $s_{abs} \leftarrow$ ABSS($s$)
39:     $a_{abs} \leftarrow$ ABSA($a$)
40:     $Cndts \leftarrow KB(s_{abs}, a_{abs})$
41:     for $i = 1 \ldots |Cndts|$ do
42:         $\hat{s}, \hat{d} \leftarrow Cndts[i]$
43:         if $\hat{s} = s'$ then
44:             $Cndts'[i] \leftarrow s', d'$
45:         else
46:             $Cndts'[i] \leftarrow \hat{s}, \hat{d}$
47:         end if
48:     end for
49:     $KB \leftarrow KB - \langle s_{abs}, a_{abs}, Cndts\rangle$
50:     $KB \leftarrow KB \cup \langle s_{abs}, a_{abs}, Cndts'\rangle$
51: end function
```

The subroutine SIBILLAADDSTATE (line 26) formalizes the behavior of adding the newly scanned GUI state to the knowledge base. SIBILLAADDSTATE executes SCAN (line 27), removes the current candidates from the knowledge base (line 28) and replaces them with either the *non-deterministic* marker, if the new state overflows the maximum number of candidates that the knowledge base allows for each entry (lines 29–30), or a list of candidates enriched with the newly scanned state (line 31–33). As explained above, setting the *non-deterministic* marker settles SIBILLAGET to always execute SCAN thereon for any subsequent query that refers to the given abstract ⟨state, action⟩ pair: for those interactions the testing technique will behave exactly as the original testing tool without SIBILLA.

SIBILLAADDDATA (line 37) updates the data associated with a GUI state. It simply refers to the abstract counterpart of the interaction represented by the GUI state s and the action a (lines 38–39), retrieves the corresponding knowledge base entry (line 40), loops through the candidate states in the entry to associate the GUI state s' with the new state data d' (lines 41–48), and set the updated candidates back in the knowledge base (lines 49–50).

2.5 Possible Sources of Imprecision

SIBILLA can occasionally result in imprecise inferences returning a guessed GUI state that does not perfectly correspond to the GUI state in the current screen. Such imprecisions are due to the best-effort approach of SIBILLA in validating the candidate states for interactions with non-deterministic outcome. After Algorithm 1, while SIBILLA enumerates the candidate GUI states that may correspond to the outcome of the last interaction according to the information in the knowledge base (Algorithm 1, line 11), it accepts the first candidate for which the testing algorithm can successfully identify a next action (Algorithm 1, line 12) that can be correctly dereferenced in the current GUI screen (Algorithm 1, line 13). This deduction is insufficient to guarantee with certainty that the inferred GUI state truly corresponds to the GUI state in the current screen, since the two states could both include the selected next action although being in fact different GUI states.

We remark that these imprecisions do not impact on the validity of the test cases that the test generator computes, since the approach of Algorithm 1 suffices to guarantee the validity of all actions executed during the testing generation process. Moreover, SIBILLA never alters the policy used by the test generator to decide the next action to execute. However, the imprecision of SIBILLA can sometime mislead the exploration. We quantify the actual imprecisions in our experiments reported in Section 3.

2.6 Abstraction Functions

As we already explained earlier in this section, SIBILLA is parametric with respect to the abstraction functions on GUI states (abstraction function $absS$) and GUI actions ($absA$) that are used. Our current prototype of SIBILLA allows engineers to plug their own abstraction functions, to specifically customize their test generation technique. It also includes three pre-defined ways of abstracting interactions, defined based on common abstractions already experienced in Web testing techniques, which differ on the degree of abstraction that they introduce:

(i) $Types$ is the configuration that introduces the strongest abstraction. In particular, $absS$ abstracts GUI states as the URL and title of their GUI page, and $absA$ abstracts GUI actions as the counting of the involved widgets grouped by type (e.g., two actions that click any button are considered equal, because they both depend on a single widget of type Button). This configuration suits well with many Web applications in which distinct interactions depend on well distinguishable pages and actions.

(ii) $Actions$ is a configuration that partially abstracts the interaction using the enabledeness abstraction, which focuses on the actions executable from each state. In particular, $absS$ abstracts GUI states as the set of enabled actions that belong to each state, and $absA$ abstracts GUI actions as the locator of each action. This kind of abstraction has been exploited by several Web testing techniques [20], [21], [22], [23].

(iii) $Widgets$ is the configuration that introduces the least abstraction, incorporating many elements of the GUI in the representation. $absS$ represents each widget in the GUI state with a subset of its properties (e.g., identifier, textual content, list of associated input fields, etc.) and $absA$ represents each action with its locator, type (e.g., enter text, click, etc.), and parameter values. This kind of abstraction has been exploited also by several Web testing techniques [3], [11], [24], [25], [26].

We use these three configurations to empirically study the tradeoffs between efficiency and precision that SIBILLA incurs when configured with different abstraction functions. We refer the reader to online material at https://gitlab.com/Sibilla/sibilla#sibilla-configurations for more details about the above abstractions.

We also considered the possibility of integrating SIBILLA with the metrics for detecting *near duplicates*, which were recently proposed for improving the effectiveness of the model inference algorithms in web testing tools [12], [27], [28]. Near duplicates are GUI states that differ from each other only by small changes that do not impact functionality, and that can thus be regarded as replicas of the same functional page. For example, a possible metric is to measure if the tree edit distance between web-page DOMs is below a given threshold. However, we found that those metrics, though well suited for model inference tasks, are not well suited for SIBILLA due to the high efficiency penalties that they incur. In Section 3.5 we provide empirical evidence of this phenomenon, and we further discuss the relation between SIBILLA and near-duplicate detection in Section 4.

We remark that SIBILLA uses the abstractions to match the interactions being executed with the ones in the key-set of the knowledge base, but not to validate if the states predicted as inferences from the knowledge base (i.e., for the matching interactions) correspond to the current ones. For the matching interactions, SIBILLA optimistically accepts the returned states as valid predictions, as long as they do not produce observable failures at the next testing step. Thus, SIBILLA can implicitly predict many near-duplicate states as the same state, if those near-duplicates are the result of the same (abstractly-matched) interaction, and are enough functionally-equivalent that they do not produce observable failures at the next testing step. Unmatched interactions lead SIBILLA to add new entries to the knowledge base. Inferences of observably-different states lead SIBILLA to record further states as non-deterministic alternatives (up to the configured n threshold).

3 EMPIRICAL EVALUATION

We studied SIBILLA in the context of Web testing techniques that use three different GUI exploration strategies, namely, the strategies Random and Q-learning as implemented in ABT [9] and the strategy Depth-first as implemented in Crawljax [3]. For each technique, we evaluated the impact of 6 possible configurations of SIBILLA, that is, SIBILLA equipped with any of the 3 abstraction functions introduced in Section 2.6, each combined with 2 possible values (1 and 5, respectively) of the parameter n (the *non-determinism tolerance capacity*), which defines how many GUI states and related state-dependent data SIBILLA can at most associate with

TABLE 1: Considered SIBILLA configurations.

Identifier	State abstraction $absS$	Action abstraction $absA$	Capacity n
Types-1	page title and URL	# widgets by type	1
Types-5	page title and URL	# widgets by type	5
Actions-1	enabled actions	action locator	1
Actions-5	enabled actions	action locator	5
Widgets-1	widgets properties	action properties	1
Widgets-5	widgets properties	action properties	5

each observed interaction before falling back to handle an interaction as non-deterministic. Table 1 summarizes the six configurations. Each configuration (column *Identifier*) corresponds to SIBILLA configured with a different combination of abstraction functions for states and actions (columns $absS$ and $absA$, respectively) and non-determinism tolerance capacity of the knowledge base (column n).

In our evaluation, the 3 Web testing techniques have been used either without SIBILLA (i.e., their original version that uses SCAN and state analysis for each GUI state, hereon the BASELINE) or with each of the 6 aforementioned configurations of SIBILLA, on 9 Web applications, repeating each experiment 10 times. In total, we executed 1,890 (3 techniques $\times 7$ configurations $\times 9$ applications $\times 10$ repetitions) experiment sessions in which the techniques were configured to perform 1,500 actions each. Each experiment session took between 1 hour and 77 hours, depending on the efficiency of the considered configurations, for about 10,000 hours of experimentation in total.

Below, we introduce the Web testing techniques and the subject applications, outline the research questions that drove our experiments, describe the experimental setting, report the results, and discuss the findings and the threats to their validity.

3.1 Web Testing Techniques

We integrated SIBILLA within the test generators Crawljax[2] and ABT, to evaluate its impact with three Web testing techniques: Depth-first exploited in Crawljax (hereon, DFS), Q-learning exploited in ABT (hereon, QLEARN), and Random that we obtained with an ad-hoc configuration of ABT (hereon, RND). The strategies behind these techniques (i.e., Depth-first, Q-learning and Random) are representative of many automatic test generators at the state of the art [3], [9], [11], [29], [30], [31]. In detail:

- DFS explores the GUI states in depth-first order, starting at the home page of the Web application under test, and backtracking at states for which all possible state transitions were already executed. To this end, the exploration algorithm maintains an internal cache of the already visited state transitions, not to be confused with the knowledge base of SIBILLA. In fact, other than being two different components, the cache of the depth-fist exploration algorithm and the knowledge base of SIBILLA differ. In all our experiments, the depth-fist exploration algorithm of Crawljax reasons with concrete states, while SIBILLA woks with abstracted state-action entries. DFS discriminates actions based on locators and states based on the edit-distance between their stringified DOM [3]. When integrated with SIBILLA, the exploration algorithm stores the incrementally computed edit-distance values as state data information associated with the GUI states maintained in the knowledge base.
- QLEARN derives test cases by alternating between exploration (i.e., executing random actions), and exploitation (i.e., executing best actions). Best actions are identified according to a Q-learning model that rewards actions based on the observable screen changes: the more the changes, the higher the reward [9], [11]. When integrated with SIBILLA, the reward values are stored in the knowledge base as state data information.
- RND selects random actions among the ones that can be executed at each GUI state. This exploration strategy does not exploit any state analysis, and thus only the GUI states are maintained in the knowledge base.

As shown in Figure 1, Crawljax is not strongly affected by the cost of GUI scans, as it saves the content of already visited states in memory for backtracking purposes. Rather, its efficiency is strongly dependent on other state analysis operations, such as the computation of the Edit Distance between DOMs to detect clones and near-duplicates while performing model inference. Other tools instead rely on systematic GUI scan operations (e.g., executed after each action), in order to capture the widgets present in the newly reached state, so to avoid runtime errors due to stale elements no longer present in the GUI. This is particularly relevant in ABT, which uses Selenium [32], a Web framework to automatically execute cross-browser tests. SIBILLA contributes to saving the time allocated to both scan and state analysis, since it saves and predicts both states and state data.

3.2 Subject Applications

We used DFS, QLEARN and RND for testing the nine subject Web applications listed in Table 2. These applications cover different domains and all have been intensively involved in several studies on Web testing (e.g., [5], [33], [34], [35], [36], [37], [38], [39]).

In our experiments, these subjects are representative of distinct types of designs that are typical for Web applications, hence challenging SIBILLA in different ways: for instance, Tricentis comes in the form of a Javascript Single-Page Application (SPA), which thus requires SIBILLA to discriminate different GUI states that correspond to the same page of the application, while Mantis Bug Tracker is a large Web app where distinct functionalities correspond to distinct Web pages, which requires SIBILLA to deal with a large corpus of states originated by different pages and functionalities. Since single Web page applications are particularly challenging for the abstractions that cannot use the current page to discriminate states, we included in our set of subjects all the single-page Web applications considered in the work of Biagiola et al. [39], with the exception of Phoenix-Trello, which is not actively maintained anymore and it was impossible to configure correctly.

The version information was not available for the Tricentis Vehicle Insurance demo since it is not an open-source project

3.3 Research Questions

Our experiments address four main research questions:

2. Release 4.1, https://github.com/crawljax/

TABLE 2: Subject Web applications.

Web App	Description	Version
DimeShift [40]	Expenses Tracking System	0.1.42
Dolibarr [10]	Enterprise Resource Planning	10.0.0
Mantis Bug Tracker [41]	Issue Tracking System	2.21.0
MRBS [42]	Event Booking System	1.9.4
Pagekit [43]	Content Management System	1.0.18
PetClinic [44]	Clinic Management Web App	11.2.11
Retrospected [45]	Projects Management System	5.1.2
SplittyPie [46]	Expenses Tracking System	1.0.0
Tricentis [47]	Vehicle Insurance Demo	-

RQ$_0$-Near Duplicate Are near-duplicate state abstraction techniques viable abstraction techniques for SIBILLA?

RQ$_1$-Efficiency What is the efficiency improvement that SIBILLA can deliver to a Web testing technique?

RQ$_2$-Deviations To what extent can the interactions executed by a Web testing technique be altered by imprecisions introduced by SIBILLA?

RQ$_3$-Determinism How many interactions are indeed deterministic (and thus foster reuse opportunities for SIBILLA) in the context of a Web testing technique?

We answer RQ_0 by using the edit distance integrated in Crawljax to compare states and achieve abstraction, comparing its execution time against the execution time of the BASELINE (i.e., normal non-optimized executions) on every Web application.

We answer RQ_1 by quantifying and comparing the total execution time of each experiment session (1,890 in total, given 3 Web testing techniques, 7 configurations, 9 Web applications, and 10 experiment repetitions).

We answer RQ_2 by quantifying and comparing, for each experiment, (i) the number of analogous steps at which a Web testing technique executes different actions when used with and without SIBILLA, respectively, (ii) the number of distinct Web form submissions that the Web testing technique exercises when used with and without SIBILLA, respectively, and (iii) the structural data about server-side code coverage of the PHP-based subjects (i.e., Dolibarr, Mantis Bug Tracker, and MRBS), representing traditional and challenging big Web applications, that can be achieved with and without SIBILLA.

We answer RQ_3 by quantifying how many interactions SIBILLA classifies as either deterministic or non-deterministic in each experiment. Given that, we also measured the percentages of GUI scan and state analysis operations that are saved by using SIBILLA with respect to the BASELINE in which GUI scan and state analysis operations are regularly executed.

3.4 Experiment Setup

All our experiments shared the same setup, consisting in running a Web testing technique until executing 1,500 actions on the GUI of the application under test. In this way, we can fairly compare execution time, explored actions, form submissions and deterministic interactions across experiments that execute the same number of actions.

In the case of QLEARN and RND, we achieved this by configuring ABT to generate 30 test cases, each consisting of 50 actions, while for DFS we simply stopped Crawljax after 1,500 actions, since the number of test cases cannot be directly controlled (rather it depends on how many times the technique backtracks to a previous state).

We experiment each Web testing technique (DFS, QLEARN and RND) in both its original version (i.e., BASELINE), which scans and analyzes the GUI states after executing each action, and in the version integrated with any of the 6 configurations of SIBILLA listed in Table 1. The considered abstraction functions correspond to the three pairs of abstraction functions that we introduced in Section 2.6.

As capacity values, we considered $n = 1$, which corresponds to the case in which SIBILLA handles only fully deterministic interactions, and $n = 5$, which lets SIBILLA tolerate a reasonable amount of non-deterministic interactions (i.e., the interactions that may result in up to 5 different GUI states). The choice of the specific value, $n = 5$, is admittedly arbitrary, but preliminary experiments indicated that this configuration sufficed for SIBILLA to handle the large majority of the non-deterministic interactions, and was thus well suited to represent the behavior of SIBILLA with handling for non-determinism. Indeed our experiments eventually confirmed that the number of interactions that exceeded the $n = 5$ non-determinism capacity of SIBILLA with $n = 5$ was at most 8%.[3]

We controlled for the impact of the random choices implemented in Crawljax and ABT in two ways. On one hand, to observe the different configurations of SIBILLA in the context of the same testing sessions, we kept the *same random seed* for each *7-experiment pass* that encompasses executing BASELINE and all 6 SIBILLA configurations, for a given Web testing technique against a given subject application. Running all configurations with a same seed allows us to compare the behavior of the techniques with and without SIBILLA at the level of the individual actions that are executed at each step of the testing process. On the other hand, to avoid the bias of specific random choices, we repeated every experiment 10 times and used different seeds for each 7-experiment pass. Furthermore, the initial state of each Web application under test was reset to the same default state before each run.

We quantified the deviations that may be due to imprecise state inference of SIBILLA with respect to (i) the specific sequences of actions that the Web testing techniques generate when using BASELINE and each configuration of SIBILLA, respectively, (ii) the number of distinct Web form submissions that they can exercise in either case, and (iii) the code coverage of the PHP-based subjects (i.e., Dolibarr, Mantis Bug Tracker, and MRBS), that BASELINE and SIBILLA can achieve.

With respect to the sequences of GUI actions, we quantified the deviations as follows. Given a subject application and a random seed, if a technique generates the sequence

3. Considering other possible configurations (n=2, n=3, n=10, ...) could be worth, but we had to compromise in not selecting too many configurations, to avoid the combinatorial explosion in the number of experiments to be executed to explore a too large set of possible options. With options considered in this paper, our experiments already amounted to over 10,000 hours of execution time.

$a_1, a_2, \ldots, a_{1500}$ when using BASELINE, and the sequence $a'_1, a'_2, \ldots, a'_{1500}$ when using a configuration of SIBILLA, we measure the deviation due to SIBILLA as the percentage of actions $a'_i \neq a_i, \forall i$. This notion of deviation is particularly strict, as it captures any deviation independently of how relevant it is for the testing activity. For instance, if the third action generated with BASELINE is generated as fourth action with SIBILLA, this is counted as a deviation.

To also measure deviations in the general testing activity, regardless of the action-to-action comparison of the test sequences, we compared the number of distinct Web form submissions exercised with BASELINE and each SIBILLA configuration across all experiments.

To consolidate our findings, we finally collected the structural data about server-side code coverage of the PHP-based Web applications (i.e., Dolibarr, Mantis Bug Tracker, and MRBS), that generally represent challenging testing subjects; for this purpose, we relied on xdebug PHP extension[4] and php-code-coverage library[5], comparing the coverage achieved by BASELINE with respect to the one achieved by SIBILLA.

In general, in our experiments the execution of the BASELINE tools represented the ground truth in terms of execution time of the testing tools, sequences of GUI interactions explored by the testing tools, submitted Web forms and accomplished coverage. We then compare how these metrics change when using SIBILLA with the different configurations, to measure efficiency gains and amounts of deviation.

3.5 Results for RQ_0: Near Duplicate

We launched the DFS test generation strategies for runs of 150 actions to explore the viability of exploiting near-duplicate abstractions to detect the similar states. We collected data about the cost of running the test generation process as is, without SIBILLA in place, and with SIBILLA configured to abstract states according to the edit distance between states. If near-duplicate state comparison runs faster than the BASELINE, it is a viable option for SIBILLA. Otherwise, if it is slower, it cannot be used to optimize the efficiency of the test generation process, regardless of the accuracy of the abstraction.

Figure 3 shows the results obtained with a sample set of four Web applications (i.e., Dolibarr, Mantis Bug Tracker, MRBS, and Tricentis), all presenting a similar pattern. The cost of performing many state comparisons to determine if a state is similar to an already discovered state reveals to be higher even than the BASELINE, which always runs the state-base analysis. In contrast, the goal of SIBILLA is improving the efficiency of the test generation process by quickly retrieving the right abstract state, and the associated information, from the knowledge base rather than retrieving the current state from the GUI. In a nutshell, near-duplicate strategies can be exploited to drive the test generation process, but cannot be feasibly used to optimize the extraction of a representation of the current GUI in SIBILLA.

4. https://xdebug.org/
5. https://github.com/sebastianbergmann/php-code-coverage

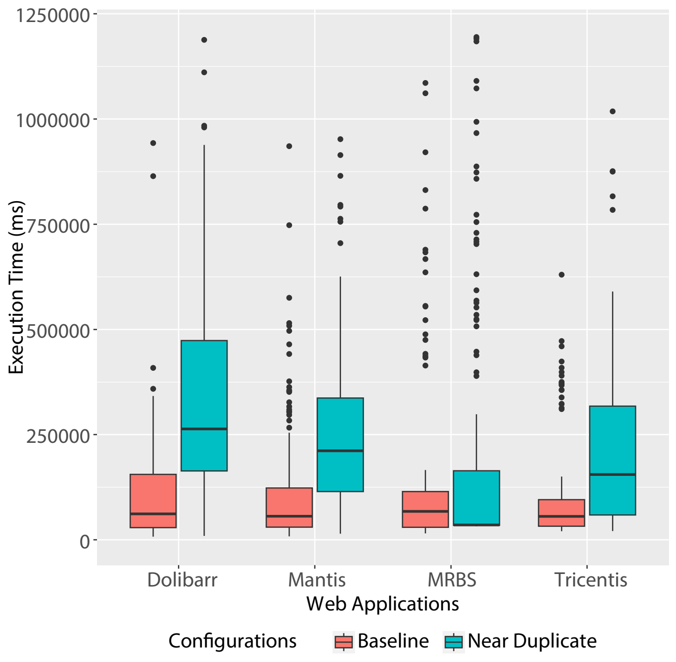


Fig. 3: A comparison of SIBILLA configured with near-duplicate and BASELINE execution time costs (milliseconds).

3.6 Results for RQ_1: Efficiency

We compared the mutual efficiency of the considered configurations of SIBILLA by measuring the time spent to complete the execution of the Web testing techniques in each experiment.

The boxplots in Figure 4 summarize the distribution of the execution time (in hours) of the 10 experiments with each subject application (as indicated at the top of each group of plots), Web testing technique (as indicated at the right of each group of plots) and configuration of SIBILLA compared with BASELINE (as indicated at the x-axis of the plot).

We observed the highest gains for Tricentis with DFS, where SIBILLA reduced the execution time of an order of magnitude with respect to BASELINE, from a median of about 47 hours to less than 2 hours in all configurations (96% of reduction). There were minimal differences between SIBILLA configurations in PetClinic, SplittyPie and Tricentis. When DFS is used, Types-5 presented optimal or nearly optimal time reduction, up to 90% (e.g., in Mantis Bug Tracker, Dolibarr, Retrospected), with the only exception of DimeShift, where the abstraction was severely challenged by some dynamics elements in the pages. Pagekit and Retrospected were the apps with the highest variance in the results. We experienced significant execution time with the lowest abstractions (i.e., Widgets-1 and Widgets-5) for Pagekit, still obtaining a median reduction of 30% with respect to BASELINE. Retrospected was the only case where SIBILLA reached an execution time higher than BASELINE for a few runs with the Actions-1 configuration. After some investigation, we found out that Crawljax struggled at exploring some of the Retrospected pages, when large sets of emojis and pop up features were present, generating incidental loops that temporarily blocked the advancement in the exploration. The same results were not observed in any run involving ABT. All SIBILLA configurations using

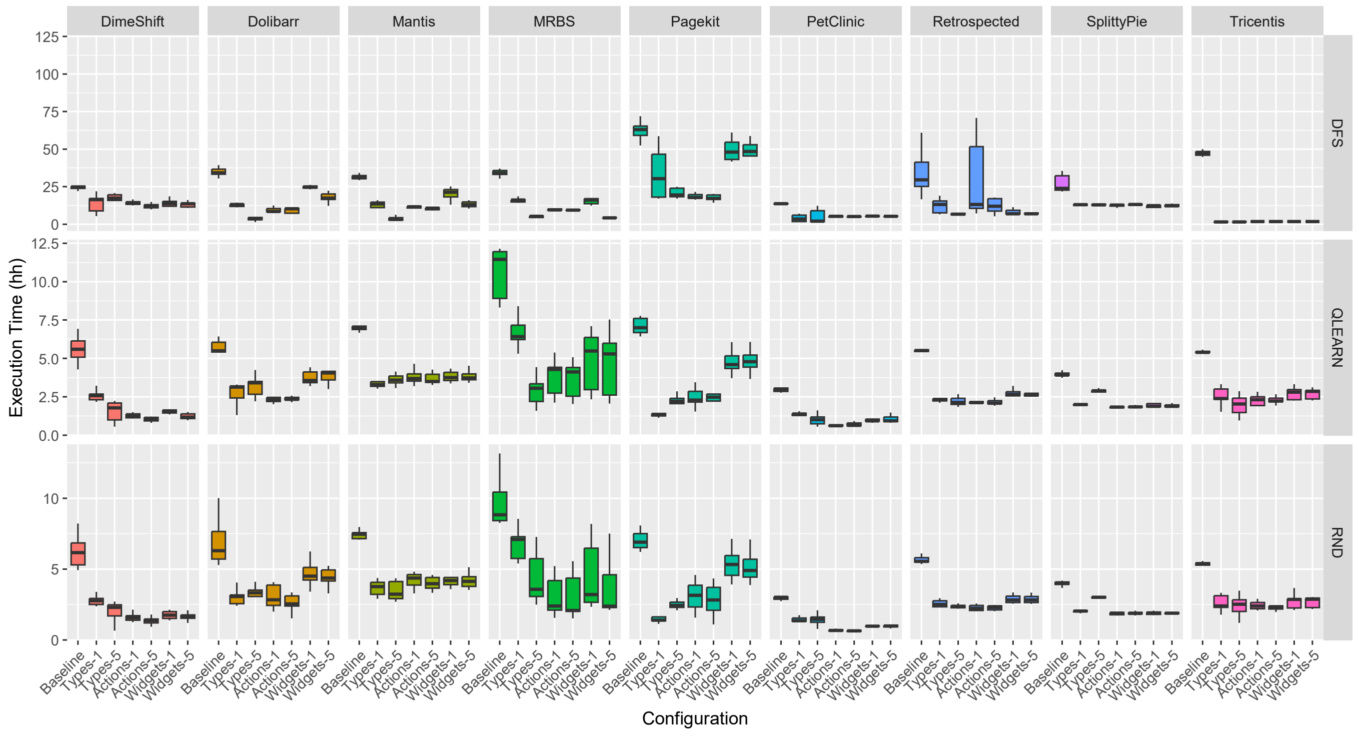


Fig. 4: Execution times (hours) of SIBILLA configurations and BASELINE.

5 capacity values have been more efficient than using 1 as capacity value, with Widgets-5 achieving the best reduction with respect to BASELINE: from about 34 hours to 4 hours (88% of reduction). These high gains confirm the benefit of using the SIBILLA knowledge base not only for GUI scanning but also for saving and reusing the result of state analysis, which is predominant in DFS (as already shown in Figure 1).

As for DFS, in RND the best SIBILLA configurations mostly used 5 as capacity value. Actions-5 resulted the best configuration for the majority of Web applications: Tricentis (with a reduction of 57% with respect to BASELINE, in the median case), Dolibarr (60% reduction), MRBS (78% reduction), PetClinic (79% reduction), DimeShift (88% reduction), and SplittyPie (53% reduction). In Mantis Bug Tracker, RND performed best with the Types-5 configuration, reducing the test time from 7.5 hours to 3.2 hours (57% reduction). Surprisingly, Types-1 resulted the best configuration in Pagekit (80% reduction). Types-1 performed the overall worst reduction in MRBS, still achieving a 22% reduction. Finally, Retrospected showed Actions-1 as the best configuration (60% of median reduction).

Also in the case of QLEARN, in most experiments the best SIBILLA configuration was one of those that used 5 as capacity value, often it was the Actions-5 configuration. Types-5 resulted the best configuration in both Tricentis (reduction of 63% w.r.t. BASELINE, from 5.4 to 2 hours in the median case) and MRBS (73%, reduction from 11.4 hours to 3.1 hours). Actions-1 and Actions-5 reached almost same time reduction in both Dolibarr (58% reduction), SplittyPie (53% reduction), and PetClinic (78% reduction). Actions-5 outperformed the other configurations in both DimeShift (81% reduction) and Retrospected (63% reduction). In contrast, for Mantis Bug Tracker and Pagekit the best configuration was Types-1, reducing execution time with respect to BASELINE by 53% and 81%, respectively. Pagekit in particular followed the same trend of RND and DFS about Widgets-1 and Widgets-5 being the most expensive configurations.

Answer to RQ1: The results indicate that SIBILLA can significantly improve the efficiency of a Web testing technique, reducing the execution time by a factor ranging between 22% and 96%, at least halving execution time in most cases. The only exception is the Retrospected application where the Actions-1 configuration experienced some slowdowns due to weaknesses in the underlying Crawljax tool that badly handles some specific Web features (e.g., pop ups). In general, for all the considered Web testing techniques and applications, all 6 SIBILLA configurations outperformed BASELINE, with the configurations using the strongest abstractions and non-determinism tolerance capacity equals to 5 performing better than the others, thanks to their capability to reuse the data in the knowledge base more often.

3.7 Results for RQ_2: Deviations

The boxplots in Figure 5 summarize the distribution of the deviations of each configuration of SIBILLA, across the 10 experiments with each technique and each subject application. All plots report the deviations in percentage with respect to executing BASELINE for same techniques and subject applications, and this is why there are not specific plots for BASELINE in the figure. Figure 5 also indicates, with diamond symbols, the percentage of distinct Web form submissions that were not exercised with SIBILLA with respect to the ones exercised with BASELINE.

The results of SIBILLA configurations were consistent in all applications and techniques.

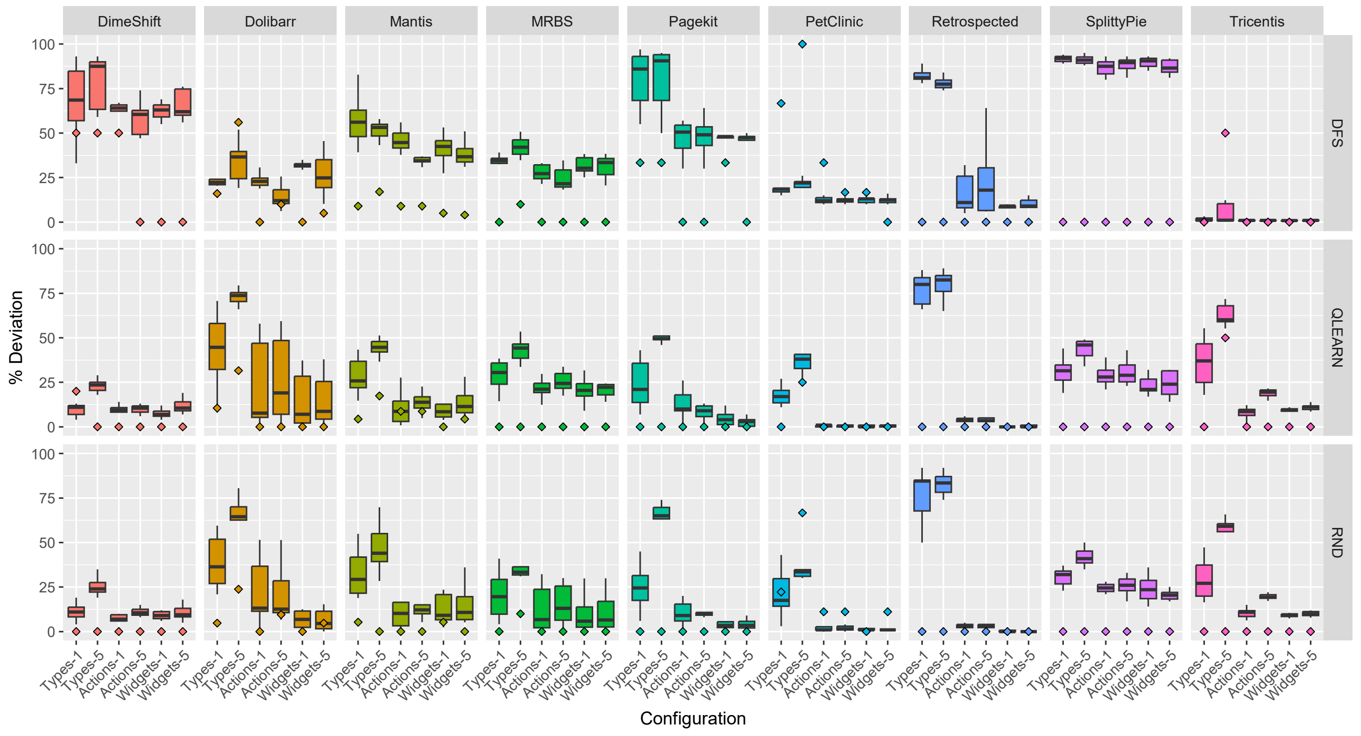


Fig. 5: Deviations (%) of SIBILLA configurations with respect to BASELINE.

In DFS, Tricentis presented the overall best outcome (percentages of deviations close to 1% in all configurations), followed by PetClinic (about 10% of deviations in all configurations but Types-1 and Types-5) and Retrospected (between 9% and 18% except for Types-1 and Types-5), since these applications are single-page and rarely affected by unexpected GUI changes that may interfere with the DFS backtracking activity. It is notable, however, how three other single-page applications (i.e., DimeShift, Pagekit, SplittyPie) behave differently, reaching between 47% and 86% median deviations in the best cases. The problem with these apps depended mostly on how Crawljax interacts with pop up menus and alerts that inhibit the interactions with the background elements, causing actions to fail silently without triggering SIBILLA. Furthermore, SplittyPie presented subtle changes in the GUI once any action occurred (e.g., by adding <div> tags to encapsulate newly entered items), so any deviations produced a potential avalanche effect, as the tool mostly relies on retrieving Web elements via XPaths.

Excluding Tricentis, where all configurations performed similarly, the configurations with the strongest abstractions performed the worst. Types-1 generated a median of 56% of deviations in Mantis Bug Tracker, 81% in Retrospected, and 92% in SplittyPie. Types-5 generated a median of 22% of deviations in PetClinic, 37% in Dolibarr, 42% in MRBS, 88% in DimeShift, and 91% in Pagekit. Actions-5 and Widgets-5 were in general the best configurations for DFS: 9% of median deviations in Retrospected (Widgets-5), 12% in Dolibarr and PetClinic (Actions-5), 22% in MRBS (Actions-5), 34% in Mantis Bug Tracker (Actions-5), 47% in Pagekit (Widgets-5), 62% in DimeShift (both Actions-5 and Widgets-5), and 86% in SplittyPie (Widgets-5).

RND and QLEARN have shown a higher variance in the results compared to DFS. This for instance is evident in Dolibarr, because of its dynamic and rich GUI. Again, for both RND and QLEARN, the worst configuration was Types-5 in all cases with the exception of Retrospected. Instead, the best performing configurations for RND and QLEARN were those implementing the least abstractions, with Widgets-1 on top (0% deviations in Retrospected with both RND and QLEARN, 1% deviations in PetClinic with both RND and QLEARN, 4% deviations in Pagekit with RND, 6% deviations in MRBS with RND, 7% deviations in Dolibarr with QLEARN, 7-8% deviations in DimeShift with both RND and QLEARN, 9% deviations in Mantis Bug Tracker and Tricentis with RND, 20% deviations in MRBS with QLEARN), followed by Widgets-5 (0% deviations in Retrospected with both RND and QLEARN and 1% deviations in PetClinic with RND, pairing Widgets-1, 3% deviations in Pagekit with QLEARN, and 21-24% deviations in SplittyPie with RND and QLEARN, respectively). Interestingly, in RND and QLEARN the issues experienced with DFS in DimeShift, Pagekit, and SplittyPie were not present. Also, Retrospected presented consistent high deviations for the stronger abstractions, as Types-1 and Types-5 were not capable of capturing the whole complexity of the GUI of the application, that is characterized by multiple actionable emojis.

The diamonds in Figure 5 show how there were no significant deviations in the number of distinct Web form submissions exercised by any SIBILLA configuration with respect to BASELINE, with the exception of those using strong abstractions. For DFS, in 7 cases out of 9, Widgets-5 was able to exercise the same forms of BASELINE with no deviations, and with very low deviations ($\leq$5%) in the other two cases (i.e., Mantis Bug Tracker, Dolibarr); Widgets-1, Actions-1, and Actions-5 behave the same as BASELINE in 6 cases out of 9, showing limited deviations in most other cases. Conversely, Types-5 was the worst configuration, deviating 50% or more times in 4 cases out of 9 (i.e., PetClinic, Dolibarr, DimeShift, Tricentis). Concerning RND, Widgets-1 and Actions-1 missed 0% of form submissions with respect to BASELINE in 8 cases out of 9 (excluding Mantis Bug Tracker and PetClinic, where they missed 5% and 11%, respectively), whereas Widgets-5 and Actions-5 reached the same results of BASELINE in 7 cases out of 9, with limited ($\leq$11%) deviations in the remaining two cases (i.e., Dolibarr and PetClinic). In QLEARN, all configurations excluding the strongest abstractions (i.e., Types-1 and Types-5) observed 0% forms missed in 8 cases out of 9, and less than 10% in the remaining case (i.e., Mantis Bug Tracker). Types-5 and Types-1 were confirmed as the worst configurations, still marking over than 25% form submissions in only 3 cases in total for Types-5 (i.e., PetClinic with RND, Tricentis and Dolibarr with QLEARN).

Figure 6 shows the boxplots of the coverages (%) regarding Actions-5, Widgets-5, and BASELINE with DFS, QLEARN, and RND techniques. After some trial executions, we observed that tracking the code coverage resulted prohibitive for longer runs, sometimes negatively affecting the cost of a run by even more than 200% (see Figure 7 of some sample runs on MRBS Web application using DFS, with and without a coverage logging tool employed, respectively), so we decided to limit these experiments to 150 actions instead of 1,500 each, and to 5 iterations instead of 10. In Figure 6, only Actions-5 and Widgets-5 are shown, as they resulted (see Figures 4-5) the most viable SIBILLA configurations both in terms of execution time and deviations.

The boxplots show limited variations, excluding MRBS. More in details, Table 3 summarizes the differences in statement coverage observed from the runs with Actions-5 and Widgets-5 of SIBILLA configurations against BASELINE. All cases except for two showed coverage differences below 5%, and half of these were below 2%. The highest differences occurred in MRBS using Widgets-5 with RND (-5.09% with respect to BASELINE, about 200 statements missing by ABT), followed by Actions-5 with DFS (-5.00%, about 400 statements missing by Crawljax). By inspecting more in details the results, we found that almost half of the differences regarded HTML tags that included some PHP code spread across multiple rows, that the coverage tool captured as different statements. Given that, SIBILLA sometimes missed statements and sometimes reached new ones, only slightly affecting the testing effectiveness for the considered cases.

Answer to RQ2: The results indicate that the configurations using the strongest abstractions (i.e., Types-1 and Types-5) result in a high number of deviations that question their viability. These configurations may indeed affect the behavior of a Web testing technique due to their imprecisions, as too high deviations may alter the exploration strategy of the testing technique, generating test cases that differ from the desired ones. Conversely, the configurations based on Actions and Widgets present low number of deviations, as well as few differences in statement coverage for the considered experiments, and negligible differences when considering how forms are exercised. These findings suggest that the least abstractions of SIBILLA can preserve the tool's exploration strategy by achieving a better execution efficiency.

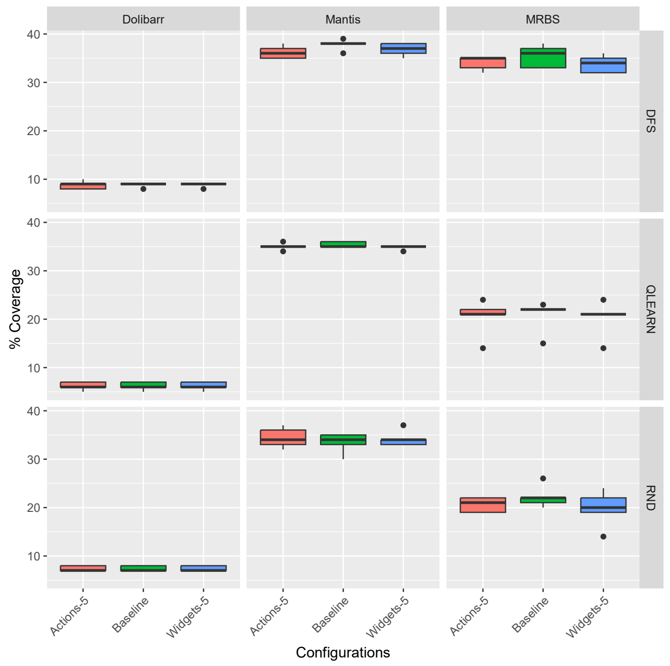


Fig. 6: Coverage % of Actions-5 and Widgets-5 configurations against BASELINE.

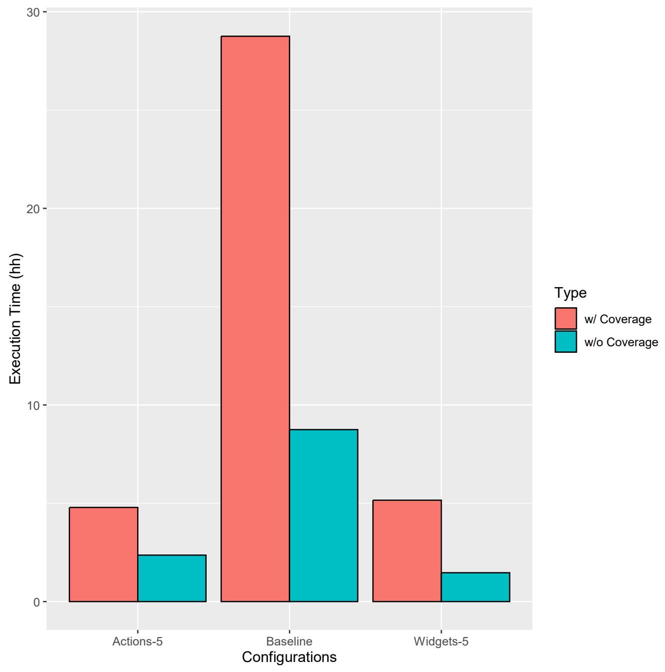


Fig. 7: Tracking coverage cost affecting Actions-5, Widgets-5, and BASELINE runs in MRBS Web application.

3.8 Results for RQ_3: Determinism

We quantified the amount of exploitable determinism by measuring the number of interactions that SIBILLA successfully associated with a finite list of GUI states in each experiment. Table 4 reports the results grouped by subject application (column *Web app*) and SIBILLA configuration (column SIBILLA *config.*), in percentage with respect to the

TABLE 3: Statements Coverage differences (%) of Actions-5 and Widgets-5 configurations against BASELINE.

	Baseline								
	Dolibarr			Mantis Bug Tracker			MRBS		
	DFS	QLEARN	RND	DFS	QLEARN	RND	DFS	QLEARN	RND
Actions-5	-0.48%	+1.67%	0.00%	-3.30%	-0.29%	+4.41%	-5.00%	-0.38%	-0.10%
Widgets-5	-0.72%	+2.57%	0.00%	-4.36%	+0.97%	+2.58%	-2.28%	-3.83%	-5.09%

TABLE 4: Deterministic and non-deterministic interactions.

Web App	SIBILLA config.	Per-interaction state set cardinality					
		1	2	3	4	5	non-det.
DimeShift	Types-1	54%	-	-	-	-	46%
	Actions-1	86%	-	-	-	-	14%
	Widgets-1	96%	-	-	-	-	4%
	Types-5	54%	19%	9%	8%	2%	8%
	Actions-5	84%	13%	1%	<1%	<1%	2%
	Widgets-5	94%	5%	1%	<1%	<1%	<1%
Dolibarr	Types-1	70%	-	-	-	-	30%
	Actions-1	95%	-	-	-	-	5%
	Widgets-1	96%	-	-	-	-	4%
	Types-5	59%	21%	8%	6%	3%	3%
	Actions-5	96%	4%	<1%	<1%	<1%	<1%
	Widgets-5	97%	3%	<1%	<1%	<1%	<1%
Mantis BT	Types-1	71%	-	-	-	-	29%
	Actions-1	94%	-	-	-	-	6%
	Widgets-1	96%	-	-	-	-	4%
	Types-5	68%	19%	6%	3%	2%	2%
	Actions-5	97%	3%	<1%	<1%	0%	0%
	Widgets-5	96%	4%	<1%	<1%	<1%	<1%
MRBS	Types-1	59%	-	-	-	-	41%
	Actions-1	94%	-	-	-	-	6%
	Widgets-1	93%	-	-	-	-	7%
	Types-5	53%	23%	11%	5%	4%	4%
	Actions-5	94%	5%	1%	<1%	<1%	<1%
	Widgets-5	94%	6%	<1%	<1%	<1%	<1%
Pagekit	Types-1	75%	-	-	-	-	25%
	Actions-1	98%	-	-	-	-	2%
	Widgets-1	99%	-	-	-	-	1%
	Types-5	70%	13%	3%	2%	9%	3%
	Actions-5	98%	2%	0%	0%	0%	<1%
	Widgets-5	99%	1%	0%	0%	0%	0%
PetClinic	Types-1	82%	-	-	-	-	18%
	Actions-1	97%	-	-	-	-	3%
	Widgets-1	98%	-	-	-	-	2%
	Types-5	68%	16%	10%	4%	1%	1%
	Actions-5	97%	3%	<1%	0%	0%	0%
	Widgets-5	98%	2%	0%	0%	0%	<1%
Retrospected	Types-1	60%	-	-	-	-	40%
	Actions-1	99%	-	-	-	-	1%
	Widgets-1	99%	-	-	-	-	1%
	Types-5	58%	38%	3%	<1%	<1%	1%
	Actions-5	99%	1%	0%	0%	0%	<1%
	Widgets-5	100%	<1%	0%	0%	0%	0%
SplittyPie	Types-1	65%	-	-	-	-	35%
	Actions-1	87%	-	-	-	-	13%
	Widgets-1	92%	-	-	-	-	8%
	Types-5	65%	21%	6%	2%	1%	5%
	Actions-5	88%	11%	<1%	<1%	0%	1%
	Widgets-5	94%	5%	<1%	<1%	<1%	1%
Tricentis	Types-1	76%	-	-	-	-	24%
	Actions-1	96%	-	-	-	-	4%
	Widgets-1	92%	-	-	-	-	8%
	Types-5	78%	12%	8%	2%	<1%	0%
	Actions-5	94%	5%	1%	<1%	0%	0%
	Widgets-5	92%	8%	<1%	<1%	0%	0%

45,000 (1,500 × 10 × 3) interactions executed in the 10 experiments with the 3 Web testing techniques, for the interactions that SIBILLA associated with exactly 1, 2, 3, 4 or 5 GUI states, or classified as non-deterministic (columns *Per-interaction state set cardinality*).

Interestingly, when the SIBILLA knowledge base allowed a non-determinism tolerance capacity up to 5 (configurations Types-5, Actions-5 and Widgets-5), SIBILLA was able to associate almost the full spectrum of observed interactions with a finite list of GUI states, assigning the classification *non-deterministic* to either no interaction (e.g., in all experiments with Tricentis) or only to negligible sets of interactions (1-8% with Types-5). Instead, when the knowledge base limited a non-determinism tolerance capacity to 1, in the worst case of Types-1 the interactions classified as non-deterministic were between 18% for PetClinic to 46% for DimeShift, with percentages below 15% in all other cases.

SIBILLA rarely observed non-determinism when the tolerance capacity was set to 5, suggesting that reuse opportunities was not limited to the strongest abstraction (i.e., Types-5). Table 5 shows the median percentages of GUI scan and state analysis operations that were saved by SIBILLA in the considered configurations and subject applications, with respect to the BASELINE in which GUI scan and state analysis were regularly executed. SIBILLA helped saving between 18% (Widgets-1, Widgets-5) and 88% (Types-5) of GUI scan and state analysis operations.

TABLE 5: GUI scan and state analysis operations avoided by SIBILLA (%).

Configuration	DFS	QLEARN	RND
Types-1	81%	46%	42%
Types-5	88%	76%	71%
Actions-1	50%	58%	48%
Actions-5	49%	62%	49%
Widgets-1	48%	23%	18%
Widgets-5	47%	24%	18%

Answer to RQ3: The results indicate that SIBILLA successfully classified most interactions as deterministic in each configuration, application, and technique, especially when the non-determinism tolerance capacity was set to 5. SIBILLA also successfully saved many GUI scan and state analysis operations, even with the least abstracting configuration. This result confirms the possibility to suitable capture the behavior of an application under test with the SIBILLA knowledge base, fostering SIBILLA reuse opportunities.

3.9 Discussion

Our results indicate some clear findings about SIBILLA. First, the strong abstractions, such as Type-1 and Type-5, may introduce too many deviations, with the risk of affecting the effectiveness of the Web testing technique, and cannot be thus advised for practical use. Instead, the abstractions based on Actions and Widgets result in acceptably low deviations, in particular when applied for testing single-page Web applications, as well as exhibiting little differences in statements coverage even when large Web applications are tested, promisingly improve the efficiency of Web testing, halving the execution time in most cases, up to improving the efficiency of Crawljax of an order of magnitude.

The results also show that for the techniques with an intrinsic degree of randomness (QLEARN and RND), SIBILLA provides stable results across the Web applications. On the contrary, deterministic techniques like DFS may show quite drastic differences in terms of efficiency and more relevant deviations. For instance, SIBILLA introduced almost no deviation and an order of magnitude improvement with DFS applied to Tricentis, while it introduced a smaller improvement and significantly more deviations for the other Web applications. This suggests that SIBILLA should be integrated with systematic techniques preferably when the subject application is likely to exhibit a mostly deterministic behavior.

It has to be noted that DFS performance was affected by pop up menus and alerts, when present in the subject applications (e.g., DimeShift, Pagekit). DFS was also affected by frequent variations in the GUI, such as those implemented by SplittyPie any time a new entry is added, as underlying Crawljax tool relies on XPath localization strategies to retrieve widgets within the GUI, that can be sometimes fragile [17], making actions to fail in high number and producing an avalanche effect in deviations from the BASELINE. These drawbacks were not observed with ABT, in the case of neither RND nor QLEARN techniques.

Finally, the results advice the use of SIBILLA with non-determinism tolerance capacity larger than 1. In our experiments, using a tolerance capacity set to 5 worked better than a list of size 1 in most cases, reducing the number of interactions classified as non-deterministic and positively impacting the efficiency of the Web testing techniques, avoiding between 18% and 88% of the GUI scan and state analysis operations.

3.10 Threats to Validity

The main threat to the validity of our conclusions is that SIBILLA may deliver its efficiency improvements, at the cost of sacrificing the effectiveness of the test cases computed by the Web testing techniques. We mitigated this threat by quantifying the deviations of SIBILLA with respect to using BASELINE, in terms of action-to-action comparisons, unexercised Web forms, and code coverage. Our results indicated a limited number of deviations for the abstractions based on Actions and Widgets, suggesting that the test artifacts produced with SIBILLA are in line with the ones produced with BASELINE.

Internal validity threats can derive from possible mistakes in our implementation of SIBILLA, in our integration of SIBILLA in Crawljax and ABT, or in the profiling functionalities that we added to collect the data that we used to answer the research questions. To avoid mistakes, we carefully inspected and extensively tested our implementations. We also made our artifacts publicly available for inspection.

The external validity threats concern the generality of our findings. We selected three different Web testing techniques (based on Depth-first, Random and Q-learning) and nine non-trivial subject applications representative of distinct types of Web GUI designs (e.g., small single-page and large traditional multi-page Web applications) involved in several past studies about Web testing. Although these cases do not cover the full spectrum of cases, they cover a number of combinations sufficient to deliver evidence of the effectiveness of SIBILLA. In fact, we remark the good consistency of our findings across our experiments (which amount to over 10,000 hours of execution time), which suggest that some configuration of SIBILLA may improve the efficiency of the Web testing techniques without affecting their effectiveness.

4 RELATED WORK

The GUI testing process is intrinsically expensive. On one hand, test generators need to explore the application under test by executing many actions before they can cover meaningful scenarios [3], [4], [11], [25], [48], [49]. Since every action that is processed generates executions that traverse many, potentially all, software layers in the application under test, actions are inevitably slow to execute compared for instance to API calls in unit test cases. On the other hand, since test generators run as independent processes, they have to actively wait for the GUI to be fully rendered before they can scan and analyze it, and finally perform an action. Repeatedly waiting for the GUI and executing scan and analysis operations further slows down the testing process, as reported in this paper.

To reduce the cost of the GUI testing process, researchers and practitioners considered different techniques and perspectives. For instance, headless browsers testing [50], [51] exploits headless browsers [52], [53], that is browsers that do not render the GUI to run tests. Headless browsers can be used to eliminate the cost of GUI rendering, speeding up interactions.

Dong et al. studied how to improve GUI exploration with the capability to save and restore application states dynamically [54]. This capability can be used to conveniently move from one state to another during exploration, without having to reset the application and replaying the actions necessary to reach a state of interest. Wen et al. [55] and Tramontana et al. [56] studied how to run GUI tests in parallel to improve the efficiency of the testing process by exploiting additional computational resources. Biagiola et al. exploited GUI models to improve the efficiency of search-based test generation for web applications [39]. Their technique measures diversity between test cases, and requires in-browser execution only for the test cases that explore diverse behaviours.

In the mobile domain, Baek and Bae [57] propose a set of multi-level GUI comparison criteria to generate accurate GUI models and improve testing effectiveness via abstraction levels. Su et al. [58] present a stochastic model-based testing approach for Android apps. Gu et al. [59] propose a gradual refinement of the GUI model abstraction that is incrementally generated at runtime.

While these techniques can improve the efficiency of GUI testing, they cannot save the cost due to the repeated execution of the scan and state analysis operations, which can be significant as reported in our experiments. SIBILLA represents a complemental solution that can be combined with other techniques to ultimately increase the efficiency of the GUI testing process.

In a recent work [60], Feng et al. recognize the problem of GUI rendering cost in the context of Mobile test automation. They propose AdaT, an image-based approach to dynamically adapt GUI rendering to test generation via real-time streaming, so as to send events until the GUI is fully rendered. The approach aims at balancing between *testing effectiveness*, negatively affected by short waits that may miss relevant events due to partially rendered GUIs, and *testing efficiency*, when long waiting times are generally imposed.

SIBILLA relies on abstractions on GUI states and GUI actions to efficiently handle the key-set of its associative knowledge base and the access to the knowledge base thereby. Other GUI testing approaches investigated abstractions, generally to improve the efficiency of inferring a suitable model of the GUI parts already visited during the executions. This problem is commonly referred to as *model inference*. GUI testing tools can benefit of model inference for several goals, including visualizing the GUI behaviors that were explored, discriminate covered and uncovered states, or implementing smart action-selection strategies, e.g., the strategies based on Q-learning of ABT [11] and Testar [26], [29]. In turn, achieving good model inference requires well tuned abstractions on the GUI states and the GUI actions represented in the models: Too fine abstractions may lead to state explosion, whereas too coarse abstractions may lead to useless models.

For example, the Testar tool has been recently extended to support parametric abstractions, allowing users to select which subset of widget properties Testar shall use to abstractly represent GUI states and actions during model inference [26], [61]. Other authors further addressed model inference by studying techniques for identifying *near-duplicate* GUI states, i.e., the GUI states that differ from each other only by small changes that do not impact functionality, and that can thus be regarded as replicas of the same functional page [12], [27], [28]. Effectively identifying near-duplicates allows GUI testing tools to infer models that are minimal in that they represent only the set of states that are indeed distinct. The study from the seminal paper of Yandrapally et al. pinpointed the tree edit distance between web-page DOMs as a good similarity metric to identify near duplicates in the context of web testing techniques [12]. Other authors investigated further similarity metrics for web-page DOMs based on applying tree-kernel functions, and establishing the equivalence after building a mapping between the fragments of the DOMs [27], [28].

Model inference and the SIBILLA's key-based access to the knowledge base can or cannot exploit the same abstractions, depending on the computation-time efficiency of those abstractions for the tasks of matching abstract states and actions. On one hand, since SIBILLA aims to directly impact the time efficiency of the testing strategy, SIBILLA can rely only on abstractions that foster very efficient matchings: ideally (modulo the possibility of deviations, as we discussed in Section 3.7) SIBILLA aims to foster a testing strategy that visits exactly the same sequences of states and actions that would be visited without SIBILLA, but significantly faster than without SIBILLA. In fact, SIBILLA cannot benefit of the state-of-the-art algorithms for identifying near duplicates, since these algorithms generally incur too high penalties for matching abstract states and actions. In Section 3.5, we provided evidence that accomplishing state matching by using the tree edit distance between web-page DOMs downgraded the performance of SIBILLA to large extent, ultimately making SIBILLA penalize (rather than improve) the efficiency of the testing tool. On the other hand, model inference can often relax the efficiency requirements, provided that the cost of the model inference algorithm, including the cost incurred for matching abstract states and actions, pays off in the ability of the inferred models to foster effective guidance during testing. Near duplicate detection has indeed demonstrated very effective for model inference tasks [12], [27], [28].

We also remark that, while in this paper we study SIBILLA with respect to a set of abstractions, we do not claim the novelty of those abstractions per se. In the future, we aim to investigate SIBILLA with further types of abstractions, e.g., the ones proposed in Testar. Moreover, SIBILLA and model inference contribute with distinct, arguably orthogonal, and possibly complementary types of improvement to the GUI testing tools, without imposing any constraint on using the same abstractions for both mechanisms. In future work, we also aim to investigate if Testar can be improved by integrating it with SIBILLA.

5 CONCLUSIONS

The efficiency of automatic Web testing is severely affected by the cost of executing GUI scan and analyzing GUI states, operations that must be repeatedly executed to iteratively extract GUI states and compute state-specific properties. To address this challenge, this paper presents SIBILLA, an approach that can effectively infer interactions that reach recurring, already analyzed GUI states. SIBILLA saves the costs of executing GUI scan and state analysis operations multiple times for those states. We reported results obtained with three Web testing techniques and nine subject Web applications that confirm that SIBILLA can deliver significant efficiency improvements: testing time is often reduced by 50%, with best cases reaching over 90% reduction, with limited deviations (using the *Actions* and *Widgets* abstractions) from the original testing algorithms.

Replication Package

The tools, results and scripts to replicate our experiments are publicly available in the following repository: https://gitlab.com/Sibilla/sibilla.

REFERENCES

[1] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip, "A framework for automated testing of Javascript Web applications," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011.

[2] A. M. Fard and A. Mesbah, "Feedback-directed exploration of Web applications to derive test models," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[3] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based Web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1, pp. 3:1–3:30, 2012.

[4] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern Web applications," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 35–53, 2012.

[5] D. Clerissi, G. Denaro, M. Mobilio, and L. Mariani, "Plug the database & play with automatic testing: Improving system testing by exploiting persistent data," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2020.

[6] UI/Application Exerciser Monkey. [Online]. Available: https://developer.android.com/studio/test/monkey

[7] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of Android test generation tools in industrial cases," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2018, pp. 738–748.

[8] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[9] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic black-box testing of interactive applications," in *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[10] Dolibarr. [Online]. Available: https://sourceforge.net/projects/dolibarr/

[11] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Automatic testing of GUI-based applications," *Software Testing, Verification & Reliability (STVR)*, vol. 24, no. 5, pp. 341–366, 2014.

[12] R. Yandrapally, A. Stocco, and A. Mesbah, "Near-duplicate detection in web app model inference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 186–197.

[13] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," in *Proceedings of the 16th International Conference on World Wide Web*, 2007, pp. 141–150.

[14] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Dependency-aware web test generation," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 175–185.

[15] A. Mesbah, A. Van Deursen, and D. Roest, "Invariant-based automatic testing of modern Web applications," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 35–53, 2011.

[16] M. Biagiola, F. Ricca, and P. Tonella, "Search based path and input data generation for web application testing," in *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2017, pp. 18–32.

[17] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Visual vs. DOM-based Web locators: An empirical study," in *International Conference on Web Engineering (ICWE)*. Springer, 2014, pp. 322–340.

[18] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 20, no. 5, pp. 522–532, 1998.

[19] M. Schur, A. Roth, and A. Zeller, "Mining workflow models from web applications," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 12, pp. 1184–1201, 2015.

[20] G. De Caso, V. Braberman, D. Garbervetsky, and S. Uchitel, "Program abstractions for behaviour validation," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[21] ——, "Enabledness-based program abstractions for behavior validation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 3, pp. 25:1–25:46, 2013.

[22] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of Mobile apps," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, 2014.

[23] O. Riganelli, S. P. Mottadelli, C. Rota, D. Micucci, and L. Mariani, "Data loss detector: Automatically revealing data loss bugs in Android apps," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[24] A. M. Memon, I. Banerjee, B. N. Nguyen, and B. Robbins, "The first decade of GUI ripping: Extensions, applications, and broader impacts." in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2013.

[25] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Transactions on Software Engineering (TSE)*, vol. 31, no. 10, pp. 884–896, 2005.

[26] T. E. J. Vos, P. Aho, F. Pastor Ricos, O. Rodriguez-Valdes, and A. Mulders, "Testar — Scriptless testing through Graphical User Interface," *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1771, 2021.

[27] A. Corazza, S. Di Martino, A. Peron, and L. L. L. Starace, "Web application testing: Using Tree Kernels to detect near-duplicate states in automated model inference," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–6.

[28] R. K. Yandrapally and A. Mesbah, "Fragment-based test generation for Web apps," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1086–1101, 2023.

[29] T. E. J. Vos, P. M. Kruse, N. Condori-Fernandez, S. Bauersfeld, and J. Wegener, "Testar: Tool support for test automation at the user interface level," *International Journal of Information System Modeling and Design (IJISMD)*, vol. 6, no. 3, pp. 46–83, 2015.

[30] G. Denaro, L. Guglielmo, L. Mariani, and O. Riganelli, "GUI testing in production: Challenges and opportunities," in *Proceedings of the Programming Workshop*, 2019.

[31] P. Aho and T. E. J. Vos, "Challenges in automated testing through Graphical User Interface," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 118–121.

[32] Selenium WebDriver. [Online]. Available: https://www.selenium.dev/documentation/webdriver/

[33] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable Web testing: An empirical assessment during test case evolution," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 272–281.

[34] M. Bures, K. Frajtak, and B. S. Ahmed, "Tapir: Automation support of exploratory testing using model reconstruction of the system under test," *IEEE Transactions on Reliability*, vol. 67, no. 2, pp. 557–580, 2018.

[35] H. Raffelt, T. Margaria, B. Steffen, and M. Merten, "Hybrid test of Web applications with Webtest," in *Proceedings of the workshop on Testing, Analysis, and Verification of Web services and applications (TAV-WEB)*, 2008, pp. 1–7.

[36] M. Hammoudi, G. Rothermel, and P. Tonella, "Why do record/replay tests of Web applications break?" in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 180–190.

[37] H. Kirinuki, H. Tanno, and K. Natsukawa, "Color: Correct locator recommender for broken test scripts using various clues in Web application," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 310–320.

[38] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella, "Web test dependency detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 154–164.

[39] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based Web test generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 142–153.

[40] DimeShift. [Online]. Available: https://github.com/jeka-kiselyov/dimeshift

[41] Mantis Bug Tracker. [Online]. Available: https://sourceforge.net/projects/mantisbt/

[42] MRBS. [Online]. Available: https://sourceforge.net/projects/mrbs/

[43] Pagekit. [Online]. Available: https://github.com/pagekit/pagekit

[44] PetClinic. [Online]. Available: https://github.com/spring-petclinic/spring-petclinic-angular

[45] Retrospected. [Online]. Available: https://github.com/antoinejaussoin/retro-board

[46] SplittyPie. [Online]. Available: https://github.com/tsubik/splittypie

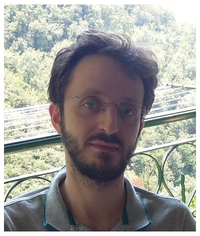
[47] Tricentis Vehicle Insurance. [Online]. Available: http://sampleapp.tricentis.com/

[48] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[49] Y. Li, Y. Ziyue, G. Yao, and C. Xiangqun, "Droidbot: A lightweight UI-guided test input generator for Android," in *Proceedings of the International Conference on Software Engineering Companion (ICSE)*, 2017.

[50] Capybara. [Online]. Available: https://github.com/teamcapybara/capybara

[51] Jasmine. [Online]. Available: https://jasmine.github.io/
[52] Mozilla Firefox. [Online]. Available: https://www.mozilla.org
[53] Google Chrome. [Online]. Available: https://www.google.com/intl/en/chrome/
[54] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of Android apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020.
[55] H. Wen, C. Lin, T. Hsieh, and C. Yang, "PATS: A parallel GUI testing framework for Android applications," in *Proceedings of the IEEE 39th Annual Computer Software and Applications Conference (COMPSAC)*, 2015.
[56] P. Tramontana, N. Amatucci, and A. R. Fasolino, "A technique for parallel GUI testing of Android applications," in *Testing Software and Systems*, V. Casola, A. D. Benedictis, and M. Rak, Eds. Springer International Publishing, 2020, pp. 169–185.
[57] Y.-M. Baek and D.-H. Bae, "Automated model-based Android GUI testing using multi-level GUI comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 238–249.
[58] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 245–256.
[59] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI testing of Android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.
[60] S. Feng, M. Xie, and C. Chen, "Efficiency matters: Speeding up automated testing with GUI rendering inference," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 906–918.
[61] A. Mulders, O. R. Valdes, F. P. Ricós, P. Aho, B. Marín, and T. E. J. Vos, "State model inference through the GUI using run-time test generation," in *Research Challenges in Information Science*, R. Guizzardi, J. Ralyté, and X. Franch, Eds. Springer International Publishing, 2022, pp. 546–563.



Diego Clerissi is a post doctoral researcher at the University of Milano-Bicocca. In 2020, he received his Ph.D. in Computer Science from the Department of Informatics, Bioengineering, Robotics, and Systems Engineering (DIBRIS) of the University of Genova, with a thesis focused on novel quality assurance approaches for Web and IoT systems. His current research interests are in software engineering, software analysis, requirements analysis, and IoT.

Giovanni Denaro is Associate Professor of computer science at the University of Milano - Bicocca. He received the Ph.D. degree in Computer Science and Engineering from Politecnico di Milano in 2002. His research interests include software testing and analysis, formal methods for software verification and cybersecurity, distributed and service-oriented systems, and software metrics. He has been investigator in several research and development projects in collaboration with leading European universities and companies. He is involved in the organization of major software engineering conferences.

Marco Mobilio is a research fellow at the University of Milano - Bicocca, where he received his Ph.D. in 2017 and his Master Degree in Computer Science in 2013. His main interests cover Software Architecture, Cloud Monitoring and Self-Healing, and Automatic Testing for web and mobile applications. He also works on Ambient Assisted Living, focusing on architectures and platforms for Human Activity Recognition in mobile environments.

Leonardo Mariani is Full Professor at the University of Milano - Bicocca. He holds a Ph.D. in Computer Science received from the same university in 2005.

His research interests include software engineering, in particular software testing, program analysis, automated debugging, specification mining, and self-healing and self-repairing systems. He has authored more than 100 papers appeared at top software engineering conferences and journals.

He has been awarded with the ERC Consolidator Grant in 2015, an ERC Proof of Concept grant in 2018, and he is currently active in several European and National projects. He is regularly involved in the organization of major software engineering conferences.