Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles

Leonardo Mariani
University of Milano-Bicocca
mariani@disco.it

Mauro Pezzè
University of Milano-Bicocca
USI Università della Svizzera italiana
mauro.pezze@usi.ch

Daniele Zuddas
USI Università della Svizzera italiana
daniele.zuddas@usi.ch

ABSTRACT

Testing software applications by interacting with their graphical user interface (GUI) is an expensive and complex process. Current automatic test case generation techniques implement explorative approaches that, although producing useful test cases, have a limited capability of covering semantically relevant interactions, thus frequently missing important testing scenarios. These techniques typically interact with the available widgets following the structure of the GUI, without any guess about the functions that are executed.

In this paper we propose Augusto, a test case generation technique that exploits a built-in knowledge of the semantics associated with popular and well-known functionalities, such as CRUD operations, to automatically generate effective test cases with automated functional oracles. Empirical results indicate that Augusto can reveal faults that cannot be revealed with state of the art techniques.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

GUI testing, automatic test case generation, semantics, oracles

ACM Reference Format:
Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. 2018. Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180162

1 INTRODUCTION

Testing software applications at the *system level* requires executing the applications through their interfaces to verify the correctness of the functionalities and stimulating all the layers and components involved in the execution. Since the number and complexity of the entities typically involved in a system-level execution could be significant, defining test cases that thoroughly sample and verify the behavior of an application is a difficult and expensive process. Automating just part of this process can dramatically improve the effectiveness of software verification activities and significantly reduce development costs, partially alleviating software developers from their verification effort.

In this paper we address the problem of automatically generating system test cases for *interactive applications*, that is, applications that interact with the users through Graphical User Interfaces (GUIs). Interactive applications (from now on simply *applications*) are commonly available in several contexts, including desktop and mobile environments, and are exploited in many domains, ranging from leisure and travel to banking and insurance.

Techniques for automatically testing interactive applications exploit structural information extracted from either the GUI or the code to generate system test cases. The techniques that analyze the structure of the GUI generate test cases that cover GUI elements based on combinatorial interaction testing and various heuristics [28, 32, 35, 43]. Those that analyze the source code instead exploit search-based and symbolic execution to generate test cases that exercise code items [18, 19, 27].

State-of-the-art techniques suffer from two relevant limitations: the *ineffective exploration of the execution space* and the *lack of oracles*. To illustrate these limitations let us consider a fault in the sign up functionality of OnShop, a demo e-commerce application available on git-hub [24]. Listing 1 shows an excerpt of the code that handles the user registration in OnShop.

```
300 private void signup() {
301   if (isValidForm()) {
302     insertIntoDB();
303     JOptionPane.showMessageDialog(SignupPanel, "Please_Login_to_get_Started!",
                "Congratulations", JOptionPane.DEFAULT_OPTION);
              ...
308     card.show(this.getParent(), "startCard");//Return to Initial Window
309   }else
310     resetForm();
311 }
315 private void insertIntoDB() {
              ...
334   if (resultSet.next()) { //User Already Exists
335     JOptionPane.showMessageDialog(SignupPanel,"Username_already_exists");
336     resetForm();
337   }
```

Listing 1: Faulty User Registration in OnShop

When a new user registers, the `signup` function is executed (line 300). If the signup form has been correctly filled in, function `isValidForm` returns true (line 301), and function `insertIntoDB` is invoked (line 302). If the username chosen by the user has been already taken by another user, this function correctly shows an error message to the user (line 335). The execution then returns to function `signup` and a message that informs the user that the registration has been completed correctly is *also shown* to the user (line 303). Finally, the application is redirected to the initial window

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180162

expecting the user to login (line 308). This fault is quite confusing for a user because the application shows both the behavior of a correct and incorrect registration in response to a single user request.

This fault *cannot* be automatically detected with state-of-the-art techniques. To reveal this fault, a testing technique has to produce a test case that performs a correct sign up twice while filling the username field always with the same value. In OnShop, this test scenario requires a sequence of at least 20 specific GUI actions to be covered. Considering the number of GUI actions that can be executed at every step of the execution, it is very unlikely that this scenario is covered with explorative approaches. Indeed, in our experiments none of the competing techniques have been able to cover this scenario (*ineffective exploration of the execution space*).

Moreover, even if this scenario is covered by chance, none of the available techniques would interpret the response of the system as a failure. The application produces an erroneous result, in terms of a wrong output message and an incorrect transition to the initial window, while state of the art solutions look for uncaught exceptions and system crashes [6], which is not the case for the OnShop sign up fault. Thus, even when the scenario is covered, no failure would be reported to the user (*lack of oracles*).

To address both the ineffective exploration of the execution space and the lack of oracles, this paper proposes *Augusto* (AUtomatic GUi Semantic Testing and Oracles), an approach that exploits *common sense knowledge* to automatically generate *semantically-relevant* test cases equipped with *functional oracles* that can reveal faults such as the one discussed above. In particular, Augusto is able to (i) cost-effectively produce test cases with useful combinations of actions only, in contrast with techniques that generate test cases with many unrelated and irrelevant actions, and (ii) detect failures that depend on the semantics of the application, in contrast with techniques revealing crash-like failures only.

Augusto relies on the intuition that there exists many popular functionalities that are implemented in similar ways and respond to a same semantics when they occur in interactive applications. Due to their popularity, the semantics of these functionalities is not typically provided explicitly since users and developers have already clear expectations. We indicate this shared expectation as *common sense knowledge*, and these functionalities as *application independent functionalities (AIF)*. Examples of AIFs are authentication operations, CRUD (Create, Read, Update, Delete) operations, and search and booking operations. These functionalities are pervasively present in software applications, and, despite minor differences, their behavior remains always the same [8, 40, 42].

On a testing perspective, AIFs represent a unique opportunity: their semantics can be specified *once for all* according to common sense knowledge, to be then *automatically adapted and reused* to test the specific AIFs present in the applications under test. In this way a relevant subset of the features present in an application (e.g., consider the number of CRUD operations that are typically present in an application) can be tested automatically, alleviating the tester from part of the verification effort. For instance, the authentication bug present in the onShop application can be *revealed* using Augusto with virtually *no effort* for the tester.

Augusto exploits the characteristics of AIFs to define an automatic testing process by introducing (i) an encoding of the semantics of AIFs with Alloy [22], which provides a flexible and powerful way to specify how a functionality affects the state of an application, (ii) a technique to discover the AIFs by analyzing the GUI of the application under test, (iii) a strategy to extract the specific semantics of AIFs and to automatically reflect the discovered information into the Alloy model, and (iv) a solution to generate effective test suites equipped with a functional oracle. Note that Augusto is not alternative but *complementary* to other automatic techniques: Augusto can *efficiently* and *effectively* test AIFs, while the rest of the functionalities can still be addressed with existing approaches.

In our evaluation, Augusto automatically recognized and effectively tested several AIFs across 7 interactive applications and revealed 7 real faults[1]. We compared Augusto to Guitar [37] and ABT [28], two representative state-of-the-art techniques, and disovered that only 2 of these faults could be revealed by the competing approaches, while the ineffective exploration of the execution space and the lack of a oracle prevented the identification of the other 5 faults. This result corroborates our hypothesis that Augusto can be significantly more effective than state-of-the-art approaches with AIFs, and that an automatic system testing process should *exploit both* Augusto, to test AIFs, and other approaches to test non-AIFs.

The paper is organized as follows. Section 2 discusses the characteristics of AIFs. Section 3 presents Augusto. Section 4 describes the empirical results. Section 5 discusses related work. Section 6 provides final remarks.

2 APPLICATION INDEPENDENT FUNCTIONALITIES

In this work we use the term *functionality* to refer to a semantically cohese and correlated *set of user operations* available on the GUI of an application, for instance, a set of CRUD operations all referring to the same entity type (e.g., money transactions). Thus a single AIF may correspond to several user operations.

Many functionalities have a consistently similar behavior that cannot be distinguished across applications, once abstracting away from concrete details. For instance, search and save operations may affect different kinds of entities, but in all cases they search and save an entity of some type. We refer to them as *application independent functionalities (AIF)*. AIFs satisfy the following properties:

- they are *commonly present* in several applications, some might be more common in certain domains, for instance the cart functionality is very common in the e-commerce domain, whereas others are generally common, such as CRUD functionalities;
- their semantics is largely *application independent* thus it can be defined abstractly in a way that is independent on the specific application. For example, the general semantics of CRUD functionalities does not depend on the type of the handled object;
- they can be activated from the GUI according to structural *GUI patterns* that users can recognize [40, 42]. For instance, the sign in and sign up functionalities in many applications use similar sets of widgets, although these widgets have different look and feel and placements in the windows.

Because of their popularity, the semantics of AIFs is part of users' *common sense knowledge*, thus they can be intuitively used without requiring special documentation and manuals.

[1] Tool and experimental material are available at http://github.com/danydunk/Augusto.

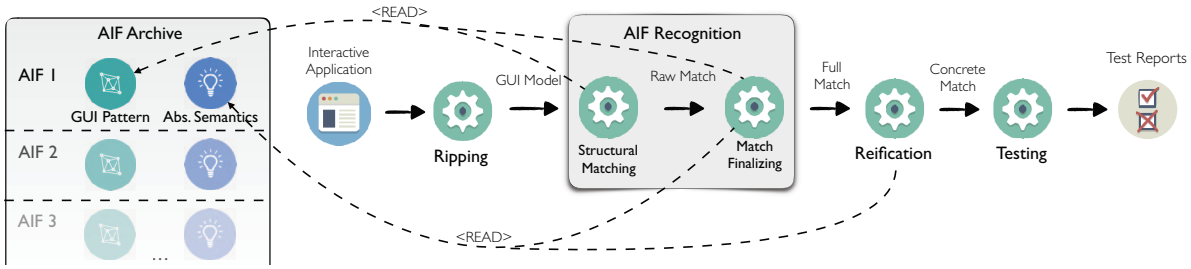

Figure 1: Augusto logical architecture

The *authentication* AIF, composed of the sign in, sing up, and sign out operations, is a good example of AIF since: (i) it provides an overall functionality, authentication, which can be found in many applications, (ii) its semantics is well-known and mostly independent on the specific application, and (iii) its presentation on the GUI is predictable and easily recognisable.

Apart from authentication, there are several other examples of AIFs: the functionality of creating, reading, updating and deleting (CRUD) objects of a type, the functionality of saving the work on a file and then reloading it, the functionality of searching and then booking a certain service (car, hotel, flight), the functionality of handling an e-commerce cart, etc. Despite their diffusion, AIFs can easily include faults, even in extremely popular applications, and thus require careful testing. For instance, faults impacting an extensive number of users have been reported for CRUD operations in Jenkins [23] and for authentication operations in Dropbox [16].

The general idea that functionalities recur in a similar way in the GUI of different applications has been already investigates in the field of UI design. There exists catalogs of UI design patterns [40, 42] and designing tools [8] that allow to create a new GUI by composing these patterns. Even if the concept of UI design pattern is not exactly the same of AIF, many UI design patterns turn out to be also AIFs.

Augusto exploits the presence of AIFs to automatically generate semantically relevant test cases equipped with functional oracles.

3 AUGUSTO: AUTOMATED AIFS TESTING

Augusto is an automatic test case generator for application independent functionalities (AIFs): it exploits the application-independent semantics of AIFs to automatically identify and test the AIFs present in interactive applications. The intuition is that, ideally, an AIF can be modelled once for all and then be exploited to effectively test any occurrence of the modeled AIF in any application. Augusto supports this intuition by offering the capabilities to discover AIFs, to automatically adapt the models to the application under test (AUT), and to generate effective test cases equipped with oracles.

To study the effectiveness of the approach, we provide an initial definition for several AIFs. Of course, the set of the defined AIFs can be further extended to increase the scope and applicability of the approach. Note that testers do not have to do any modeling effort because they can benefit from the AIF definitions already present in the tool.

In the rest of this section, we first provide an overview of Augusto and then discuss the individual elements and steps of the approach.

3.1 Overview

Figure 1 shows the logical architecture of Augusto. The *AIF Archive* is the repository that contains the set of AIFs supported by Augusto. Each AIF is modelled as a pair <*GUI Pattern*, *Abstract Semantics*>, where the GUI Pattern specifies the set of windows and widgets that may refer to the AIF (Section 3.2), and the Abstract Semantics specifies the behavior of the AIF (Section 3.3).

Augusto works in five steps. The *Ripping* step executes the application under test to dynamically extract the GUI model, which is a partial model of the structure of the GUI (Section 3.4). The *Structural Matching* step exploits the GUI Model to identify the AIFs, by searching for instances of the GUI Patterns in the GUI Model (Section 3.5). This step produces a set of raw matches, which can be partial, that is, only a subset of a GUI Pattern might match the GUI Model. Augusto supports partial matches because the GUI Model extracted through ripping might be incomplete. The *Match Finalizing* step generates additional executions aiming to complete the partial matches while verifying the consistency between the behaviors specified in the Abstract Semantics model and the behavior of the application (Section 3.6). This step produces a set of full matches, which includes every AIFs that have been fully matched in terms of its GUI pattern and its abstract semantics. The *Reification* step further refines the full matches extracting properties about the concrete behavior of the application (Section 3.7). For instance, every CRUD operation may include a different number of unique and mandatory fields for the creation of an entity. Augusto extracts these properties by stimulating the application with different combinations of inputs. This step produces a set of concrete matches, each being an AIF that occurs in the application under test. The concrete matches are associated with semantics information that takes into consideration the specific characteristics of the application under test. Finally, the *Testing* step generates and executes test cases that both combine multiple operations in a semantically relevant way and include a functional oracle to check the correctness of the results produced by the application (Section 3.8).

3.2 GUI Pattern Model

The GUI pattern model specifies how a certain AIF generally occurs on the GUI of interactive applications, and it is used by Augusto to automatically recognize whether the AUT implements the AIF. Even though there exists powerful UI modelling languages such as IFML [12], these languages are meant to model the concrete UI of a specific application, and they are not meant to model abstract portions of UIs that must be general and flexible and that can fit

multiple applications. For this reason we defined an ad hoc language for the GUI pattern model.

The language we defined for the GUI Pattern models specifies how AIFs occur in GUIs as sets of *abstract windows* that contain *abstract widgets* and are connected through *abstract edges*.

An *abstract window* identifies windows of the application. It is defined as a set of abstract widgets that are required to be present in the window. An *abstract widget* abstractly refers to a widget in the GUI, and might be of type (i) *action*, which represents widgets that can be clicked, for instance buttons, (ii) *input*, which represents widgets that can be used to enter data, for instance text fields, and (iii) *selectable*, which represents widgets that can be selected, for instance lists or tables. Abstract widgets are annotated with both regular expressions, which specify the labels that must be associated with the widgets, and cardinality, which expresses the quantity of that particular widget that can be in a window and can be either one (exactly 1), some (1 or more), none (no occurrences), lone (1 or 0) or any (0 or more).

```
<window id="loginform" card=one>
      <action_widget id="signup" card=lone>
            <label>^(register|signup|sign up).*$</label>
      </action_widget>
      <action_widget id="login" card=one>
            <label>^(login|enter|sign in).*$</label>
      </action_widget>
      <input_widget id="pass" card=one>
            <label>^(pass|password).*</label>
      </input_widget>
      <input_widget id="user" card=one>
            <label>^(user|username|email).*</label>
      </input_widget>
</window>
<window id="signupform" card=one>
      <action_widget id="register" card=one>
            <label>^(ok|save|record|signup|sign up)</label>
      </action_widget>
      <input_widget id="signupuser" card=one>
            <label>^(user|username|email).*</label>
      </input_widget>
      <input_widget id="signuppass" card=one>
            <label>^(?!re-enter|repeat)(pass|password).*</label>
      </input_widget>
      <input_widget id="signuppass2" card=lone>
            <label>^(repeat|re-enter|confirm).*</label>
      </input_widget>
      <input_widget id="otherfields" card=any>
            <label>.*</label>
      </input_widget>
</window>
<window id="loggedpage" card=some>
      <action_widget id="logout" card=one>
            <label>^(logout|exit|sign out|signout).*$</label>
      </action_widget>
</window>
<edge type=uncond from=signup to=signupform/>
<edge type=uncond from=logout to=loginform/>
<edge type=cond from=register to=loginform;loggedpage/>
<edge type=cond from=login to=loggedpage/>
```

Figure 2: AUTH GUI Pattern model

Figure 2 shows a simplified GUI pattern for the authentication AIF (for the complete model see http://github.com/danydunk/Augusto). The pattern is defined in xml format. The `window` xml elements define the abstract windows that correspond to the windows of the application. For instance, the `login` abstract window corresponds to the presence of a window that includes an input field for the username, an input field for the password, an action widget to login, and an optional action widget for registering. The definitions are flexible. They are not bound to specific GUI widgets, for instance buttons, but refer to general classes of widgets, for instance

```
/* GUI elements definition */
sig loginform, signupform, loggedpage extends Window{}
sig login, signup, register, logout extends Action_widget{}
sig user, pass, pass2,..., otherfields extends Input_widget{}
one sig Curr_win { /* Current window */
  is_in: Window one -> Time,
}
/* Functionality internal state elements */
sig Usr {
  username: one Value,
  password: one Value
}
sig Users{
  list: Usr set -> Time
}
/* Semantic Property */
one sig Required{
  fields: set otherfields
}
pred preconditions [w: Widget, t: Time] {
  w in register ⟹ not user.content.t=none ∧ not pass.content.t=none ∧
    (∀ us: Users.list.t | user.content.t≠us.username) ∧
    pass.content.t=pass2 ∧ (∀ iw: Required.fields | not iw.content.t=none)
}
pred postconditions [w: Widget, t,t': Time] {
  w in register ⟹ one us: Users | us.username=user.content.t ∧
    us.password=pass.content.t ∧ Users.list.t'=Users.list.t+us ∧
    (Curr_win.is_in.t'=loginform ∨ Curr_win.is_in.t'=loggedpage)
}
```

Figure 3: AUTH Abstract Semantics model

action widgets, and allow elements to be optional, for instance the register action widget in the `login` window. The flexibility in the definition of the cardinality is also useful for abstraction, for example the cardinality of the `otherfields` field in the `signupform` abstract window allows the pattern to match an arbitrary number of fields. The pattern definition allows for additional elements, that is, a window matching an abstract window may include elements not specified in the pattern.

Abstract edges connect an action widget of an abstract window to another abstract window to indicate possible execution flows. *Unconditional* abstract edges indicate that the target window is always reached when interacting with the source action widget, for example clicking on a navigation menu. *Conditional* abstract edges indicate that the target window is reached only if certain preconditions are satisfied, for instance successfully submitting a form. The definition in Figure 2 uses two conditional and two unconditional edges. Uncertainty is represented as a list of possible target windows. For example the edge associated with the *register* action widget indicates that once registered the execution may reach either the welcome page (automatic login) or the login form.

Abstract windows are logical windows, thus a same concrete window of an application may host multiple logical windows, for instance the login and registration abstract windows might be found in a same concrete window. Windows may have a cardinality to indicate that they are not required to be present in the target application. This might be useful for example in cases like confirmation windows which might or might not be shown in an application.

3.3 Abstract Semantics Model

The *Abstract Semantics* model describes the behavior of an AIF, and formally specifies the effect on the application of the interactions with the widgets defined in the corresponding GUI Pattern in terms of: (i) the condition necessary to successfully execute an operation

(precondition); (ii) the window that is shown after the execution of an action (transition); (iii) the state of the application after the execution of the action (postcondition). Augusto uses the abstract semantics model to generate test cases and oracles.

We specify the Abstract Semantics model using the Alloy specification language [22], chosen because of both its simplicity and expressiveness, and the efficiency of the Alloy Analyzer, an automatic tool able to analyze an Alloy model and simulate the execution of the operations defined in the model.

Figure 3 shows an excerpt of the Abstract Semantics model of the authentication AIF whose pattern model is shown in Figure 2 (for the complete model see http://github.com/danydunk/Augusto). The model declares the windows and the widgets relevant to the specified functionality (lines 2–4). The widgets defined in the GUI pattern are annotated with a tag (not shown in the example) whose value is the identifier of the corresponding widget in the Alloy model. In this way, after mapping a GUI Pattern to the concrete GUI of the application, every action on a widget can be associated with its semantics expressed in Alloy. Then the model defines the state variables that are necessary to define the behavior of the functionality (lines 5–15). In the figure, the model defines the current window (lines 5–7) and the list of registered users (lines 9–15).

Finally the model defines the preconditions (lines 20–24) and the postconditions (lines 25–29) of the operations. The figure shows pre and postconditions only for the registration operation. The precondition requires the username (`user`) and the password (`pass`) to be not empty, the repeated password (`pass2`) to be the same than the password, all the required fields (`Required.fields`) to be not empty, and the username to be unique. The postcondition adds a new user to the set of registered users and changes the current window to either the `loginForm` or the `loggedPage` window. For simplicity we omitted some of the checks in the precondition, such as the individual validity checks on the input fields.

The behavior of an AIF can be specified only partially, since it may depend on some specific *semantic properties* that change from application to application. Augusto can enrich the model by automatically plugging-in semantic properties inferred during the Reification step. To support semantic properties, the model specifies in advance one or more items that might be affected by a property that will be fully defined at a later stage. In a sense, the model must be ready to incorporate the properties that are dynamically extracted by running the application under test. In the model in Figure 3, the item `Required`, which expresses the concept of some fields *required* to be filled in to submit the registration form, is an example of a property that is indicated in advance simply as a set of fields (from line 17 to line 19) and that is refined based on the interaction with the actual application. We discuss the supported properties and the strategy to infer them in Section 3.7.

3.4 Ripping

The *Ripping* step produces a graph that represents the structure of the GUI of the interactive application in input, following the GUI ripping technique defined by Memon et al. [32]. Augusto creates the graph by recursively clicking on all the widgets in the GUI according to a depth first strategy and creating a node for every traversed window and an edge for every observed transition between windows. Augusto annotates the nodes with detailed information about all the widgets displayed in the windows.

Ripping may not be able to discover all the edges and windows. In particular, it might be unable to traverse some conditional edges because it might fail in satisfying the precondition of the functionality associated with the edge. Augusto addresses this incompleteness when recognizing AIFs in the next steps of the process.

3.5 Structural Matching

The *Structural Matching* step searches for *raw matches* between the AIFs defined in the AIF archive and the GUI model produced in the ripping phase. In particular, a raw match is a *subgraph* of the GUI Model (i.e., a subset of its windows and edges) that includes all the elements of a GUI Pattern that can be discovered through ripping.

More rigorously, a window w in the GUI model (i.e., a node of the graph) matches an abstract window aw if there exists a matching widget in w for each abstract widget in aw. A widget matches an abstract widget if the widget is of the type defined in the abstract widget (either action, input or selectable) and its label is accepted by the regular expression defined in the abstract widget. When the label is not on the widget itself, the label is identified by searching for a descriptor placed nearby the widget according to the algorithm defined by Becce et al. [7]. The matching between a window and an abstract window considers the cardinality of the widgets. The left part of Figure 4 shows a match between the definition of the `loginform` abstract window and the `Sign In` window of OnShop.

Since the GUI model extracted through ripping does not include conditional edges, the structural matching considers only the unconditional edges defined in the GUI pattern. In practice, Augusto finds a raw match if it recognizes all the windows reachable by navigating the unconditional edges of the GUI pattern in the GUI model. The conditional edges, if present in the pattern, are searched in the next step.

For example, the portion of GUI relevant to the authentication pattern discovered through ripping in the `onShop` application corresponds to the two windows shown in Figure 4 inside the grey frame. These windows are the windows reachable by navigating unconditional edges only (unconditional edges are shown with a green thick line in Figure 4). These two windows correspond to two of the abstract windows that compose the authentication pattern reported in Figure 2, thus generating a raw match between the GUI model and the AUTH pattern.

In general, the problem of identifying GUI patterns in the GUI model is an instance of the subgraph isomorphism problem, which is proven to be NP-complete [15]. However, since the number of distinct windows in an application is commonly low, the problem can be solved in few seconds, as confirmed in our empirical experience.

3.6 Match Finalizing

The *Match Finalizing* step aims to complete the raw matches, that is, each raw match is either discarded or extended to a full match by including the conditional edges.

For each conditional edge to be confirmed, Augusto generates a probing GUI interaction that samples the edge. A probing GUI interaction is a test case that terminates with an execution of the conditional edge in the AUT when its precondition is satisfied. If

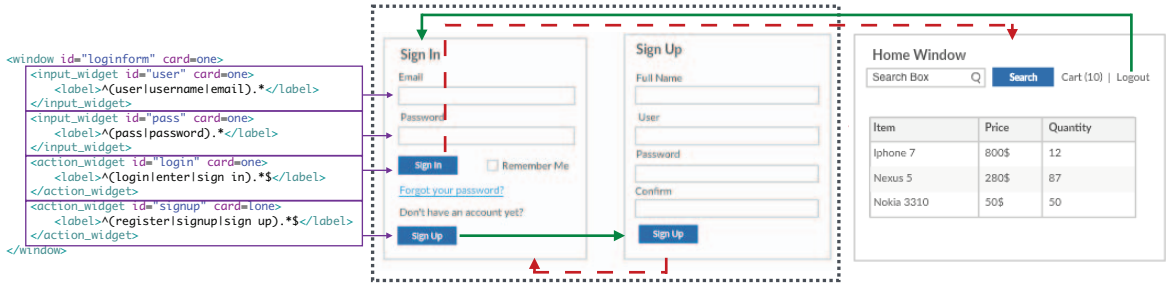

Figure 4: A simplified version of the OnShop GUI. The windows in the grey frame are those that are discovered by the ripping. Dashed red edges are discovered during the match finalizing step.

its execution reaches the expected window and satisfies the post-condition associated with the edge in the Abstract Semantics model, Augusto confirms the presence of the conditional edge, and adds the edge, as well as any newly discovered window, to the GUI model. If Augusto succeeds in confirming every conditional edge relevant to the pattern that originated the raw match, it transforms the raw match into a full match, otherwise it discards the raw match.

In some cases, a conditional edge may have more than one possible resulting window, such as for the conditional edge associated with the *register* widget defined in Figure 2. According to the pattern after a registration has been successfully completed, a program is expected to reach either the login or the welcome (abstract) window. In these cases Augusto expects a consistent behavior from the application, that is, when successfully executing a conditional edge it expects the application to always reach the same window.

Augusto generates the probing GUI interactions exploiting the Alloy Analyzer, which can be instructed to generate a sequence of GUI actions that covers a certain operation or condition of the Alloy model. The Alloy Analyzer requires in input the abstract semantics model, a condition that must be covered, and the maximum length of the interaction sequence that must be produced. In this case, the Alloy Analyzer is asked to generate sequences, of length up to a given boundary, that execute the patterns conditional edges.

If the tested AIF is not available in the initial window of the AUT, Augusto analyzes the GUI model to find the shortest sequence of actions that reaches the window with the AIF from the initial window, and adds this sequence as a prefix of the probing GUI interaction generated with the Alloy Analyzer.

When executing a GUI interaction that requires input values, such as filling a textfield, Augusto uses an archive of input values organized according to their type (e.g., emails are distinguished from dates) and divided between valid and invalid values. The archive includes predefined values for most common data types, but it can be extended with values specific for an AUT.

In the case of the sample raw match of the AUTH pattern with the onShop application, Augusto successfully generates probing GUI interactions that confirm the two conditional edges present in the pattern, shown with dashed red arrows in Figure 4. This also leads to the identification of a new window and finally turned the raw match into a full match.

3.7 Reification

The *Reification* step adapts a full match to the specific semantics of the application, by focusing on the *semantic properties* defined in the Abstract Semantics model. The Abstract Semantics model encodes the semantic properties in a general way, that is, semantic properties may have unspecified parts that are automatically adapted to the specific characteristics of the AUT. For instance, the property that requires some fields to be non-empty is defined in Figure 3 as being associated with a set of input widgets, but the exact set of widgets is left unspecified. The Reification step adapts the semantic properties to the behavior observed for the AUT.

Augusto starts by generating a probing GUI interaction that covers the behavior affected by a semantic property, and exploits the result of the execution to guess the semantic property. For example, a probing GUI interaction may try to execute the `Sign Up` operation present in the `Sign Up` window of Figure 4 with a non-empty `Full Name`, being `Full Name` the only field that needs to be determined as required or not. In fact fields `username`, `password` and `repeated password` are known to be required (see Figure 3). After executing this probing GUI interaction, Augusto, using the Alloy Analyzer constraint solver, makes a guess consistent with the collected evidence. For instance it may guess that the field `Full Name` is mandatory. Augusto automatically includes the guess in the Alloy model by adding some fields to the set of fields affected by the property –in this example it adds `Full Name` to `Required.fields`– and tries to generate a new probing GUI interaction that violates the newly guessed semantic property. The new interaction can either confirm or refute the guess. If the interaction refutes the guess, Augusto makes a new guess based on the newly collected evidence. This process iterates until either there is only one possible guess consistent with all the collected observations or a timeout is reached. In both cases Augusto incorporates the guess in the model. In the example, the first guess is correct and it is confirmed by an interaction that fails to sign up with an empty `Full Name`.

This process is quite general and can discover several classes of semantic properties. The current version of Augusto supports any semantic property that can be expressed as a property associated with a (possibly empty) set of elements of the GUI, for instance the property that an input field in a form is either *required* or *unique*.

3.8 Testing

The *testing* phase generates test cases that stimulate the discovered AIFs within semantically relevant usage scenarios. In particular, Augusto generates a test suite that satisfies the following criteria.

Conditional edge coverage. This criterion requires sampling the AIFs in every execution context: for each condition associated with a conditional edge of the model, and for each combination of truth

values computed according to MC/DC [20], there must exist a test case that exercises that combination. We selected MC/DC because it offers a good compromise between cost and completeness.

Pairwise edge coverage. This criterion requires combining the execution of multiple edges to test combinations of actions. For each ordered pair of edges in the concrete match, there must exist a test case that exercises the pair. If an edge is a conditional edge, it must be executed twice, with a satisfied and a violated precondition.

Augusto generates GUI test cases that satisfy these criteria using the Alloy Analyzer in the same way as it generates the probing GUI interactions of the previous steps. Note that the generated test cases cover the semantics of the operations by construction.

In addition, Augusto generates a functional oracle for each test case by mapping the postconditions, which define the window that must be displayed after the execution of a GUI action and its content, into assertions that are checked after the execution of every action.

Let us consider our running example. In order to cover the conditional edge about the registration operation with MC/DC (see line 22 of Figure 3), Augusto generates a non-trivial test case that first registers a new user and then registers again a user with the same username of the already existing user. The test case is also equipped with a functional oracle that checks that the current window is still the window with the registration form, after an error message has been possibly displayed. The execution of the test causes a failure detected by the oracle because the onShop application, in addition to showing an error message, behaves like if the registration has been completed successfully, which violates the generated oracle.

4 EMPIRICAL EVALUATION

Our empirical evaluation addresses 3 research questions:

(RQ1) How effective is Augusto in *detecting* application independent functionalities?

This research question investigates the capability of Augusto to automatically detect the presence of the modelled AIFs in the tested applications.

(RQ2) How effective is Augusto in *testing* application independent functionalities?

This research question investigates Augusto's ability to automatically generate test cases and find faults in the detected AIFs.

(RQ3) How does Augusto *compare* to state of the art testing techniques in testing AIFs?

This research question investigates if testing the AIFs present in an application with Augusto delivers better results than testing the same functionalities with other approaches, thus motivating the adoption of Augusto in addition to existing techniques. We used the GUITAR [37] and ABT [29] testing techniques for the comparison.

To answer these research questions we developed a prototype of Augusto for Java desktop applications. For the purpose of the evaluation, we populated the AIF archive with the definition of three AIFs: CRUD, that is adding, removing, updating and deleting objects of a type; AUTH, that is signing up, signing in and signing out from applications; and SAVE, that is saving data in files and loading them. We produced these definitions before identifying the subject applications. These AIFs are modelled according to the *common sense* knowledge by the authors of this paper.

For our empirical study, we selected as subjects seven interactive applications from different application domains, five of which were already used in previous studies [5, 28, 29]: Buddi v3.4.0.8 [13], a personal finance and budgeting program; UPM v1.6 [39], a password manager; Rachota 2.3 [26], personal tasks and activities management application; TimeSlotTracker v1.3.1 [9], another personal tasks manager application; PDF-sam v0.7 [41], a tool for merging and splitting PDFs; OnShop [24], a demo e-commerce application available on git-hub; and Spark v2.7.5 [21], a LAN chat client. Since a database is required to enable all the functionalities in Buddi and UPM, we configured an initial db with custom data for Buddi and an empty db for UPM.

The three techniques compared in RQ3 required the same configurations, that is, a pool of input values that can be used during the testing activity and the definitions of some configuration parameters. For all the techniques, we populated the pool of inputs value with the same valid and invalid values, defined coherently with the nature of the data processed by the subject applications.

In our evaluation, we used the best configuration possible for each tool, based on our knowledge of the techniques. In Augusto, we used a test case length of 15 GUI actions for all applications with the exception of OnShop that has been tested with a test case length of 22 actions. We set to 30 minutes the maximum amount of time for the reification step. In ABT we used episodes of 30 actions (note that since each episode can start from any state of the system, the resulting test cases can have an arbitrary length) and the $\epsilon-$greedy policy with $\epsilon = 0.8$, as used in ABT original paper [28]. In all the experiments ABT has been executed for the same time than Augusto. Finally, for GUITAR we generated the test cases using the EFG model and 3-wise coverage for test case generation, which guarantees GUITAR to be executed for a longer time (in some case significantly longer) than Augusto, thus favouring GUITAR over Augusto. Notice that we tried to use GUITAR also with other types of models [5, 43], but we failed since the tool always produced corrupted test suites despite our best effort (including the attempt to receive support from the developers of the tool).

Since GUITAR and ABT are not limited to AIFs, simply running the tools on the full applications would produce incomparable data for RQ3. We know by construction that GUITAR and ABT can test applications more broadly than Augusto and any result obtained by these tools with non-AIFs could not be achieved with Augusto. The purpose of RQ3 is to investigate if the opposite is also true, that is, if Augusto can deliver better results than competing approaches when testing AIFs. Only for the purpose of RQ3, to make this comparison as fair as possible and have GUITAR and ABT spending all the time testing AIFs only, as Augusto does, we modified the subject applications disabling every functionality that is not an AIF.

Finally, to mitigate the randomness in the results, we repeated all the experiments three times and reported average values.

4.1 RQ1 - AIF Detection

To answer RQ1, we studied the completeness and precision of the algorithm for detecting AIFs. We first identified the AIFs actually present in the subject applications by opening and inspecting every window of every application looking for instances of the three defined AIFs (CRUD, AUTH, SAVE). We identified a total of 17

Table 1: RQ1 - AIF Detection

AUT	AIF	ID	Match	Structure	Sem. Properties	
					Compl.	FP
UPM	CRUD	1	yes	precise	100%	0
	SAVE	2	yes	precise	n/a	n/a
Spark	AUTH	3	(yes)	precise	100%	0
Rachota	CRUD	4	yes	precise	100%	0
		5	yes	precise	100%	0.7
		6	no	-	-	-
OnShop	AUTH	7	yes	precise	100%	1.0
Buddi	CRUD	8	yes	lack delete button	100%	0.7
		9	yes	precise	100%	0
		10	yes	precise	100%	0
		11	(yes)	precise	50%	3.7
		12	yes	precise	100%	0
	SAVE	13	yes	lack replace file window	n/a	n/a
PDFsam	CRUD	14	(yes)	precise	100%	0
TTracker	CRUD	15	yes	precise	100%	0
	CRUD	16	no	-	-	-
	CRUD	17	no	-	-	-

occurrences across the applications. Note that an AIF occurrence is the occurrence of the set of operations specified in the AIF. For example, an instance of a CRUD includes operations to create, read, update and delete the entities of a kind. The applications and their AIFs are reported in the *AUT* and *AIF* columns of Table 1, respectively. Each AIF is associated with an identifier (column *ID*).

We then executed Augusto on the applications and checked the discovered matches. We indicate the result of this check in column *Match*: *yes* corresponds to the generation of a concrete match that can be used for generating test cases, *no* indicates that no match is found, and *(yes)* means that the match required manual intervention to be found. Out of 17 cases, Augusto missed only 3 AIFs. For TTracker the missed matches are caused by the limitation of the *ripping* phase that was not able to discover the GUI portions that contain the AIFs. The missed AIF in Rachota was caused by two CRUD AIFs sharing some windows, a case not supported by Augusto. Augusto never identified a non-AIF functionality as an AIF, that is, it never produced false positives during AIF detection.

Augusto required manual intervention to deal with cases not supported by the prototype in 3 of the 14 identified AIFs. In the case of Buddi (case 11), we manually excluded a Combo Box producing behaviors that are not supported by our technique. To address cases 3 and 14 we extended the definition of two GUI Patterns to accept labels that are not typically used for the operations of CRUD and AUTH. For instance, we set the label `accounts` as a valid alternative of `sign up/register` in AUTH. Although these are small interventions, they prevented the fully automatic execution of the approach in three cases.

We also evaluated the accuracy of the discovered matches in terms of the widgets included in the AIF match: Column *Structure* indicates if the match includes *all and only* the widgets that we manually identified as related to the AIF. The value *precise* indicates a perfect match, that is, no missing neither unrelated widgets associated with the AIF. Note that in 12 out of 14 cases Augusto produced a perfect match. In case 8 Augusto missed only an element, reported in the table, due to particular implementation choices in the application, and in case 13 Augusto missed a window because of a bug in the application (the bug was then reported in the testing phase). In no case Augusto associated unrelated widgets to the AIF, that is, Augusto never confused the additional elements present in a window with the ones that refer to the identified AIF.

Table 2: RQ2 - Effectiveness

AUT	AIF	ID	Avg TC	Avg Fail	Avg FA	Avg Fault	#Fault (Crash)
UPM	CRUD	1	17.7	6.7	0.3	2.0	3 (1)
	Save	2	75.7	1.0	0.7	0.3	1 (1)
Spark	Auth	3	33.7	6.7	6.7	0	0 (0)
Rachota	CRUD	4	8.3	0.7	0.7	0	0 (0)
		5	76.0	7.3	7.3	0	0 (0)
OnShop	Auth	7	17.0	4.5	4.0	0.3	1 (0)
Buddi	CRUD	8	17.0	5.5	5.5	0	0 (0)
		9	18.0	2.7	2.7	0	0 (0)
		10	18.7	0	0	0	0 (0)
		11	22.7	12.7	6.3	1.0	1 (0)
		12	19.3	0	0	0	0 (0)
	Save	13	50.7	12.3	0	1.0	1 (0)
PDFsam	CRUD	14	9.4	0	0	0	0 (0)
TTracker	CRUD	15	11.7	0	0	0	0 (0)
						Overall	7 (2)

We also evaluated the ability of Augusto to identify semantic properties, in this case to identify the required and unique fields for CRUD and AUTH AIFs. We evaluated this aspect by considering completeness, defined as the percentage of required and unique fields identified correctly by Augusto (column *Compl.*), and false positives, defined as the average number of fields wrongly associated with a required or unique property (column *FP*). We report the value *n/a* when the AIF does not include any semantic property to be discovered.

The results obtained with semantic properties show that Augusto is quite effective both in terms of completeness, only in one case some fields have not been associated with the corresponding property, and rate of false positives, only in four cases there are false positives. Note that completeness and the number of false positives associated with semantic properties could be improved by allocating more time to the reification phase.

In a nutshell, Augusto has been able to identify the AIFs present in the subject applications in 82% of the cases (in 3 cases requiring a manual intervention) producing highly accurate matches, including 86% perfect matches. Moreover, it has been able to identify the vast majority of the semantic properties present in the application.

4.2 RQ2 - Effectiveness

The effectiveness of testing techniques is typically assessed considering code coverage and their fault revealing ability. Since Augusto does not target the whole application, code coverage metrics are not informative. Thus, to answer RQ2 we evaluated Augusto considering its fault revealing ability. In particular, we measure the number of faults revealed in the subject applications.

Table 2 reports for each AIF identified by Augusto, the average number of generated test cases (column *Avg TC*), the average number of test cases that fail because of the violation of a functional oracle (column *Avg Fail*), the average number of false alarms produced, that is, the number of failing test cases that do not expose any fault in the program (column *Avg FA*), the average number of faults detected per AIF in a run (column *Avg Fault*), and the total number of faults detected in the three runs (column *#Fault*). Column *#Faults* also indicates the number of faults that cause program

Table 3: RQ3 - Comparison

AUT	Time (h)	Augusto	ABT		GUITAR	
			Reported	Covered	Reported	Covered
UPM	3.0	4	2	1	1	1
Spark	2.0	0	0	0	0	0
Rachota	2.5	0	0	0	0	0
OnShop	8.0	1	0	0	0	0
Buddi	11.0	2	0	0	0	0
PDFsam	1.5	0	0	0	0	0
TTracker	1.5	0	0	0	0	0
Overall Reported		7	2		1	

crashes. The classification of a failing test case as fault revealing or as false alarm was performed manually by this paper's authors.

The average number of test cases generated by Augusto varies a lot, ranging from 8.3 to 76.0. This big variability, which might be observed even for AIFs of the same kind in the same application (e.g., see number of test cases for the CRUDs in Rachota), depends on the specific structural match, concrete semantics and semantic properties that are extracted. This shows how Augusto, although it uses a built-in semantics for the AIFs, is able to flexibly adapt these definitions to the specific case, generating a number of test cases that depends on the actual complexity of the tested functionality.

Augusto may produce false alarms, as reported in the table. This is due to two main reasons: acceptable mismatches between the semantics model and the concrete behavior of the application, and imprecise semantics properties inference. Both these sources of imprecision cause the generation of an imprecise functional oracle. Note that in several cases sets of failures refer to a same cause (e.g., a single imprecise property may cause the failure of multiple test cases) and identifying the cause of the failure for one test can be used to drastically reduce the inspection time of the other tests failing for the same reason.

In the evaluation, Augusto has been able to reveal a total of 7 faults, with only two faults causing program crashes. This result shows that the automatic functional oracle included in the test cases is an essential element for revealing failures beyond crashes.

Augusto revealed some interesting faults, such as the one described in the introduction of this paper. Another interesting fault was detected in UPM: When editing the identifier of an account, if the change is undo and the account is saved, the operation fails with an error message stating that the identifier already exists, even though the identifier is the current identifier of the edited account.

In a nutshell, Augusto has been able to generate a number of test cases for the AIFs present in several applications and revealed multiple faults, including several non crashing faults.

4.3 RQ3 - Comparison

Table 3 shows the results obtained by Augusto, ABT and GUITAR when testing AIFs. Column *SUT* indicates the subject application. Column *Time* reports the time spent by Augusto to test the application. ABT has been executed for the same amount of time, while GUITAR has been configurated to be executed *at least* for that time. Column *Augusto* indicates the number of faults detected by Augusto. For ABT and GUITAR the table distinguishes between reported and covered faults. A reported fault is a crashing fault revealed by ABT or GUITAR (ABT and GUITAR do not include a functional oracle and can only reveal crashing faults). A covered fault is a fault that has been activated by a generated test case, but no failure has been reported due to the lack of an oracle.

All the faults reported and covered by ABT and GUITAR are a subset of the faults reported by Augusto, confirming the higher effectiveness of semantics approaches when testing AIFs. Augusto has been able to test interesting cases and interesting combinations of actions revealing 7 faults, while for 4 of these faults ABT and GUITAR have not been even able to produce the sequence that covers the faulty case. Moreover, even when ABT or GUITAR manage to cover the fault, there is a good chance that the fault is not reported due to lack of non-trivial oracles. In our evaluation, together ABT and GUITAR reported 2 crashing faults and covered but did not report another fault.

Finally, notice that Augusto computation time is compatible with server-side quality assurance sessions as well as with overnight usage of the technique. Augusto main performance bottleneck is the constraint solving performed by the Alloy Analyzer to generate test cases. This aspect might be potentially improved employing a formula caching framework to reduce the need of constraint solving and thus speeding up the technique [3, 4].

In a nutshell, compared to other state of the art techniques, Augusto has been able to sample the execution space of the AIFs more effectively and to report failures that could not be reported by the competing approaches, at the cost of reporting some false alarms. Augusto proved to be an effective complement to current *general purpose* GUI testing techniques.

Limitations. Augusto most obvious limitation is that it can be exploited to test only AIFs and cannot be used to test an arbitrary functionality, while other approaches could in principle be exploited to test any kind of operation, although their effectiveness depends on the complexity of the tested operations.

In addition, Augusto depends on the AIF archive, which assumes that the GUI of the tested application follows common sense, while in practice people might do choices against common sense. Moreover, the patterns exploit labels, which makes the archive sensitive to the language of the tested application and to the choice of terms. This limitation can be mitigated defining multiple patterns for different languages and/or using automatic strategies to find synonyms in a specific context, as done in the work by Mariani et al [30].

Threats to validity. A threat to internal validity is the generality of the AIFs models that we used in our evaluation. To mitigate the risk of defining models that fit the applications used in the evaluation but not others, we defined the AIF archive before selecting the subject applications.

Another threat to internal validity is related to the manual activities performed by the authors to classify the failing test cases reported by Augusto as *faulty* or *false alarm*, and to modify the subject applications for RQ3. For the first threat, to reduce any bias, only the failing test cases for which all the authors agree that they expose a fault were classified as faulty. For the second threat, after modifying the applications we verified that the AIFs continue working the same including the presence of the faulty behaviors.

The external validity threats of our study relate to the generality of the results with respect to the set of AIFs and set of applications

that we used. Although we cannot make claims about the generalizability of the results to other AIFs, the AIFs that we used were all successfully matched and have been all useful to reveal faults. We thus expect Augusto to be able to effectively exploit other AIFs too.

In terms of subject applications, to mitigate any issue with generalizability, we selected applications that belong to a variety of domains, most of which were already used in other studies, which facilitates comparison, and experimented with a relatively high number of AIFs per application.

5 RELATED WORK

Automation has been investigated extensively in software testing [2, 31]. Techniques for the automatic generation of system test cases have focused on two classes of complementary approaches: techniques that sample the execution space according to a model derived from the GUI of the AUT, and techniques that sample the execution space according to a model derived from the source code.

The techniques that use a model extracted from a GUI sample the execution space according to a coverage criterion defined on the model, such as covering every GUI action or every pair of dependent GUI actions [1, 5, 32–35, 43]. These approaches can uniformly sample the portion of the execution space represented in the model but provide no guarantee on the semantic relevance of the generate tests. On the contrary, Augusto includes mechanisms to complete the initial GUI model and directly generate test cases that cover semantically meaningful scenarios, thus avoiding to waste time and resources on testing irrelevant scenarios.

Instead of generating the model and generating the test cases in two sequential steps, ABT uses Q-Learning to build the model while generating system test cases, alternating exploration and exploitation activity [28, 29]. Although the test generation strategy is different, ABT still generates test cases that may cover scenarios that are relatively relevant on a testing perspective. As reported and discussed in this paper, Augusto can be dramatically more effective than these approaches in the domain of AIFs.

Other techniques exploit the AUT source code to apply symbolic execution or search-based algorithms to test case generation [18, 19]. Although these techniques may cover meaningful testing scenarios in the attempt to cover code statements, they are still limited in their ability to capture the semantics of a program and might hardly scale to complex GUIs and large programs. Augusto overcomes both problems since it exploits semantics information and does not depend on the source code.

More in general, *none of these test case generation techniques can reveal failures that do not cause crashes*, which is a key ability of Augusto, as reported in this paper.

The need of moving from explorative approaches mainly using structural information to a different class of approaches that can directly address the semantics of the AUT is also supported by studies such as the one by Choudhary et al. [14]. The study shows that, even though there exist elaborated techniques that use complex structural information, the most effective testing technique for Android applications is still a technique that simply performs random clicks on the GUI. We interpret this result as a clear evidence of the ineffectiveness of automatic testing techniques if they are not guided by semantic information and as a motivation for this work.

Other researchers approached the problem of generating semantically relevant executions in a complementary, although related, situation, that is, generating complex and semantically relevant input data for testing [10, 11, 30, 38]. In particular, Link [30] can exploit semantic Web technologies to generate sets of coherent and semantically relevant input values to execute forms. These solutions could be used to populate Augusto's input values archive.

Previous works partially investigated the use of patterns to facilitate testing [17, 36, 44, 44]. Ermuth et al. proposed a technique to infer *macro-events*, that is, GUI operations composed of several low-level GUI events (e.g., open drop down menu and click on a menu item) from usage traces. Differently from AIFs, macro-events are application specific and do not include information on how they may affect the application state. Zaeem et al. introduced the concept of *user-interaction features*, that is, sequences of operations without input-data that both have little impact on the application state (e.g., double-screen-rotation and pause-and-resume) and have a known effect. Compared to AIFs, user-interaction features are rather simple, do not need to be discovered from the GUI, and have a semantics that does not require adaptation. Moreira et al. instead exploited *UI design patterns*. This approach shares some ideas with Augusto, although Augusto has several unique capabilities: the automatic detection of known AIFs in a GUI, the automatic adaptation of AIFs definitions to the actual semantics of the application, and the automatic generation of test cases equipped with oracles.

Finally, Augusto is not the only technique that uses Alloy to generate test cases. For instance, TestEra [25] can generate test cases for Java methods from pre-post conditions written in Alloy.

6 CONCLUSIONS

This paper presents Augusto, a GUI test case generation technique that can automatically produce system test cases for application-independent functionalities (AIF) that: (i) systematically cover semantically relevant scenarios and (ii) include precise functional oracles that can reveal non-crashing faults. To obtain this result, Augusto encodes the high-level commonly expected semantics of AIFs into models that are automatically adapted to the specific characteristics of the application under test.

Our empirical evaluation shows that Augusto can precisely identify AIFs and then generate complex test cases able to exercise and report real non-crashing failures that cannot be detected with other state of the art techniques. Indeed, of the 7 faults discovered by Augusto only 2 could be reported by the competing approaches.

Our evaluation also shows that AIF models are quite resilient to the minor differences that might occur between the different implementations of a same AIF across different applications. For instance, in the evaluation our AIF models required minor changes only in 3 cases.

ACKNOWLEDGMENTS

This work is supported by the Swiss National Science Foundation with the project "ASysT: Automatic System Testing" (grant n. 200021_162409) and by the H2020 "Learn" project funded under the ERC Consolidator Grant 2014 program (ERC Grant Agreement n. 646867).

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE '12)*. ACM, 258–261.

[2] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[3] Andrea Aquino, Francesco A. Bianchi, Meixian Chen, Giovanni Denaro, and Mauro Pezzè. 2015. Reusing Constraint Proofs in Program Analysis. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '15)*. ACM, 305–315.

[4] Andrea Aquino, Giovanni Denaro, and Mauro Pezzè. 2017. Heuristically Matching Solution Spaces of Arithmetic Formulas to Efficiently Reuse Solutions. In *Proceedings of the International Conference on Software Engineering (ICSE '17)*. IEEE Computer Society, 427–437.

[5] Stephan Arlt, Andreas Podelski, Clement Bertolini, Martin Schaf, Indradip Banerjee, and Atif M Memon. 2012. Lightweight static analysis for GUI testing. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '12)*. IEEE Computer Society, 301–310.

[6] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[7] Giovanni Becce, Leonardo Mariani, Oliviero Riganelli, and Mauro Santoro. 2012. Extracting Widget Descriptions from GUIs. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE '12)*. Springer, 347–361.

[8] Roland Bennett. Patternry. http://patternry.com/patterns/. (Accessed: 2017-08-12).

[9] Roland Bennett. TimeTracker. https://sourceforge.net/projects/ttracker/. (Accessed: 2017-08-12).

[10] Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems* 5, 3 (2009), 1–22.

[11] Mustafa Bozkurt and Mark Harman. 2011. Automatically generating realistic test input from web services. In *Proceedings of the International Symposium on Service Oriented System Engineering (SOSE '11)*. IEEE Computer Society, 13–24.

[12] Marco Brambilla and Piero Fraternali. 2014. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann.

[13] Buddi. The Digital Cave. http://buddi.digitalcave.ca. (Accessed: 2017-08-12).

[14] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?(E). In *Proceedings of the International Conference on Automated Software Engineering (ASE '16)*. IEEE Computer Society, 429–440.

[15] Stephen A. Cook. 1971. The Complexity of Theorem-proving Procedures. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC '71)*. ACM, 151–158.

[16] Dropbox. Yesterday's Authentication Bug. https://blogs.dropbox.com/dropbox/2011/06/yesterdays-authentication-bug/. (Accessed: 2017-08-12).

[17] Markus Ermuth and Michael Pradel. 2016. Monkey see, monkey do: Effective generation of GUI tests with inferred macro events. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '16)*. ACM, 82–93.

[18] Svetoslav Ganov, Chip Killmar, Sarfraz Khurshid, and Dewayne E Perry. 2009. Event listener analysis and symbolic execution for testing GUI applications. In *Formal Methods and Software Engineering*. Springer, 69–87.

[19] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. Search-based system testing: high coverage, no false alarms. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '12)*. ACM, 67–77.

[20] Kelly J Hayhurst and Dan S Veerhusen. 2001. A practical approach to modified condition/decision coverage. In *20th DASC. 20th Digital Avionics Systems Conference*. NASA Langley Technical Report Server, 1B2/1–1B2/10 vol.1.

[21] Igniterealtime. Spark. https://igniterealtime.org/projects/spark. (Accessed: 2017-08-12).

[22] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology* 11, 2 (2002), 256–290.

[23] Jenkins. ISSUE 25012. https://issues.jenkins-ci.org/browse/JENKINS-25012?jql=issuetype. (Accessed: 2017-08-12).

[24] Himalay Joriwal. OnlineShopping. https://github.com/himalayjor/OnlineShoppingGUI/tree/master/OnlineShopping. (Accessed: 2017-08-12).

[25] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2011. TestEra: A Tool for Testing Java Programs Using Alloy Specifications. In *Proceedings of the International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, 608–611.

[26] Jiri Kovalsky. http://rachota.sourceforge.net/en/index.html. (Accessed: 2017-08-12).

[27] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '16)*. ACM, 94–105.

[28] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2012. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, 81–90.

[29] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2014. Automatic testing of GUI-based applications. *Software Testing, Verification and Reliability* 24, 5 (2014), 341–366.

[30] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2014. Link: Exploiting the Web of Data to Generate Test Inputs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, 373–384.

[31] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. 2015. Recent Advances in Automatic Black-Box Testing. In *Advances in Computers*. Elsevier.

[32] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *Proceedings of The Working Conference on Reverse Engineering (WCRE '03)*. IEEE Computer Society, 260–269.

[33] Atif M. Memon, Ishan Banerjee, Bao Nguyen, and Bryan Robbins. 2013. The First Decade of GUI Ripping: Extensions, Applications, and Broader Impacts. In *Proceedings of The Working Conference on Reverse Engineering (WCRE '13)*. IEEE Computer Society, 11–20.

[34] Atif M. Memon and Qing Xie. 2005. Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software. *IEEE Transactions on Software Engineering* 31, 10 (2005), 884–896.

[35] Ali Mesbah, Engin Bozdag, and Arie van Deursen. 2008. Crawling AJAX by Inferring User Interface State Changes. In *Proceedings of the International Conference on Web Engineering (ICWE '08)*. ACM, 122–134.

[36] Rodrigo MLM Moreira, Ana CR Paiva, and Atif Memon. 2013. A pattern-based approach for GUI modeling and testing. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '13)*. IEEE Computer Society, 288–297.

[37] Bao N Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering* 21, 1 (2014), 65–105.

[38] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. 2012. Automated Discovery of Valid Test Strings from the Web Using Dynamic Regular Expressions Collation and Natural Language Processing. In *Proceedings of the International Conference on Quality Software (QSIC '12)*. IEEE Computer Society, 79–88.

[39] Adrian Smith. Universal Password Manager. http://upm.sourceforge.net/index.html. (Accessed: 2017-08-12).

[40] Jenifer Tidwell. 2010. *Designing interfaces: Patterns for effective interaction design*. "O'Reilly Media, Inc.".

[41] Andrea Vacondio. PDFsam. https://sourceforge.net/projects/pdfsam/. (Accessed: 2017-08-12).

[42] Martijn van Welie. Pattern library. http://www.welie.com/patterns/index.php. (Accessed: 2017-08-12).

[43] Xun Yuan, Myra B Cohen, and Atif M Memon. 2011. GUI Interaction Testing: Incorporating Event Context. *IEEE Transactions on Software Engineering* 37, 4 (2011), 559–574.

[44] Razieh Nokhbeh Zaeem, Mukul R Prasad, and Sarfraz Khurshid. 2014. Automated generation of oracles for testing user-interaction features of mobile apps. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '14)*. IEEE Computer Society, 183–192.