Received 22 December 2022, accepted 20 March 2023, date of publication 28 March 2023, date of current version 3 April 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3262746

RESEARCH ARTICLE

AdapTV: A Model-Based Test Adaptation Approach for End-to-End User Interface Testing of Smart TVs

MOHAMMAD YUSAF AZIMI[1], CELAL CAGIN ELGUN[2], ATIL FIRAT[2], FERHAT ERATA[3], (Member, IEEE), AND CEMAL YILMAZ[1], (Member, IEEE)
[1]Faculty of Engineering and Natural Sciences, Sabanci University, 34956 Istanbul, Turkey
[2]Çayırova Central R&D Directorate, Arçelik, 34944 Istanbul, Turkey
[3]Department of Computer Science, Yale University, New Haven, CT 06520, USA

Corresponding author: Mohammad Yusaf Azimi (azimi@sabanciuniv.edu)

This work was supported in part by UNIT Information Technologies Research and Development Ltd., Turkey, through the Information Technology for European Advancement (ITEA) eXcellence In Variant Testing (XIVT) Project; and in part by the Scientific and Technological Research Council of Turkey under Project 9180066. [1]

ABSTRACT We introduce a model-based feedback-driven test adaptation approach for end-to-end user interface testing of smart TVs. From the perspective of the TV software, the proposed approach is a non-intrusive and completely black-box approach, which operates by interpreting the screen images. Given a test suite, which is known to work in an older version of the TV, and a new version of the TV, to which the test suite should be adapted, the proposed approach first automatically discovers user interface models for both the older and the new version of TV by opportunistically crawling the TVs. Then, each test case in the test suite is executed on the old version, and the path traversed by the test case in the respective UI model is found. Finally, a semantically equivalent path in the UI model discovered for the new version of the TV is determined and dynamically executed on the new version in a feedback-driven manner. We empirically evaluate the proposed approach in a setup that closely mimics the industrial setup used by a large consumer electronics company. While the proposed approach successfully adapted all the test cases, the alternative approaches used in the experiments could not adapt any of them.

INDEX TERMS Consumer electronics testing, model-based testing, smart TV testing, test adaptation.

I. INTRODUCTION

Today's smart TVs have large and complex codebases, which, as any other software system, need to be tested thoroughly. However, as smart TVs are quite different than mobile and/or general-purpose computing platforms, testing them has its own challenges. One of these challenges, which is also the focus of this work, is the need to automatically adapt the existing test cases to the new version of the TV. This is an issue of great practical importance, especially for the end-to-end user interface (UI) testing of these systems, since the TV user interfaces tend to change in every version of the TV, which typically breaks many (if not all) of the existing tests.

The associate editor coordinating the review of this manuscript and approving it for publication was Vivek Kumar Sehgal.

One company that suffers from the fragility of their UI test cases is Arcelik - the fourth largest home appliances manufacturer in Europe operating in 100 different countries under 10 different brand names, including Beko and Grundig. Arcelik specifies the end-to-end UI test cases for their smart TVs as a sequence of low-level remote controller (RC) commands, such as pressing the `left`, `right`, `up`, `down`, or `OK` button on the RC. To execute a test case, the commands are fed to the TV one after another and the outcomes, i.e., whether the test executions were successful or not, are checked by analyzing the screenshots taken from the TV as needed. Further information regarding this industrial setup is provided in Section V-E.

Arcelik is, indeed, maintaining hundreds of UI test cases in this form, encoding decades of experience in testing smart TVs. Although abstractions are extensively used in the

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

development of these test cases in the sense that the commonalities between the test cases are captured in the modules of their own, changes in the UIs (which typically occur at almost every new version of the TV released to the market) break almost all of the existing test cases. An overly simplified hypothetical example would be: a test case consisting of the RC commands `[down, right, right, OK]` to open up the sound menu in the older version of the TV, may suddenly become `[up, left, left, OK]` in the new version. Needless to say, manually adapting the broken test cases is tedious, error-prone, and costly [2], [3], [4], thus not sustainable at all.

It, however, turns out that automating the test adaptation process in the TV domain (and in the consumer electronics domain in general) has its own specific challenges on top of all the other challenges of test adaptation in the related domains, including the mobile and web application domains [5], [6], [7]. One challenge is that, although an integral part of test adaptation is to figure out what is actually on the screen, smart TVs, unlike the web and mobile platforms, do not necessarily provide object models for their screens, such as Page Object Models (POMs) or Document Object Models (DOMs), which report many useful attributes for the UI elements present on the screens, including their types, locations, and labels [8], [9], [10], [11]. Therefore, in the TV domain, the screens need to be automatically interpreted by using artificial intelligence (AI), including image processing and optical character recognition (OCR).

Another challenge is that the changes in TV user interfaces typically result in profound changes in the sequence of interactions to be carried out (especially in the higher levels of the menu hierarchy), thus the sequence of screens to be visited, to get the same functionality out of the TV. Most of the existing test adaptation approaches, on the other hand, tend to assume that changes are often confined within individual screens, such that the sequence of screens to be traversed stays intact. When this assumption does not hold, the issues encountered during the adaptation process, are typically resolved in an ad hoc manner by, for example, carrying out random or semi-random interactions with the hope of reaching the targeted UI elements and/or screens. This, however, adversely affects both the effectiveness and the efficiency of the test adaptation process in the TV domain where even going from one UI element to another element on the very same screen, may require a sequence of well-planned interactions to be carried out. For example, since there is (in the general case) no mouse or touch screen present for the interactions, reaching a targeted UI element on the screen requires pressing a correct sequence of buttons on the RC.

To address these challenges, we, in this work, present a feedback-driven model-based test adaptation approach, called *AdapTV*. At a very high level, given an older (S) and new version (S') of the TV and a test suite (T), which is known to work on S, AdapTV 1) automatically discovers the UI models M and M' for S and S', respectively, by opportunistically crawling the user interfaces; 2) for each test case in T, executes the test case on S and finds the path traversed in M; and 3) determines a "semantically equivalent path" in M' and dynamically executes the path in a feedback-driven manner on S'.

From the perspective of the TV software, the proposed approach is a non-intrusive and completely black-box approach, thus addressing our first challenge discussed above. This is because the proposed approach operates by interpreting the screenshots of the user interfaces. And, the rationale behind using a model-based approach is to minimize the guesswork in the presence of UI changes, which, in turn, can improve both the effectiveness and the efficiency of the test adaptation processes, thus addressing our second challenge. This is because having a UI model not only helps get a big picture of the user interface, so that all the UI elements located in other parts of the UI, can be figured out, but it can also help construct a plan to reach any of these UI elements by finding a path in the model.

In this work, we are, in particular, concerned with the UI test cases, which aim to test the UI states of a TV, rather than the internal states. To this end, Arcelik uses two main types of test oracles. One type of oracle checks to see if an expected sequence of UI states is visited. Another type checks to see if the destination (i.e., the final) UI state has been reached or not, regardless of the states visited in between. Therefore, we define two different adaptation criteria, namely *route equivalency* and *destination equivalency*, implementing the former and the latter types of test oracles, respectively.

To evaluate the proposed approach, we carry out empirical studies in a setup, which closely mimics the industrial setup used by Arcelik. In the experiments, we use the real test cases developed by Arcelik, which reflect the decades of experience Arcelik has gained in testing smart TVs, together with some automatically generated test cases, which are designed to increase the diversity in testing. While the proposed approach successfully adapted all the test cases under both adaptation criteria, the alternative approaches used in the experiments could not adapt any of them under any criterion.

In our previous work (an industrial abstract) [12], we briefly introduced the idea and the rationale behind our model-based adaptation approach without providing any further details and without reporting any empirical results. In this work, however, we provide all the details regarding the proposed approach and empirically evaluate it by using real as well as randomly generated test cases.

Note that although we have evaluated the proposed approach on the TVs produced by a single company, namely Arcelik. The proposed approach can readily be applied to test other TVs as well as other consumer electronics with screen-based UIs. This is mainly due to the fact that, from the perspective of the system software, we treat the system as a black box and provide a non-intrusive adaptation approach.

The remainder of the paper is organized as follows: Section II introduces the industrial case; Section III provides a motivating example; Section IV presents the proposed approach; Section V carries out the empirical studies; Section VI discusses the threats to validity; Section VII presents the related work, and Section VIII finishes with the concluding remarks and some future work ideas.

II. INDUSTRIAL CASE

Arcelik expresses the UI test cases for its smart TVs in terms of low-level RC commands. One reason for this is to carry out the end-to-end testing of the TVs in a completely black box manner. Another is that, for various technical reasons, including certain security and privacy concerns, the TV does not provide any facilities for automating testing at the system level.

As an example, Figure 1a presents the home screen taken from Grundig TV v02.025.00. An overly simplified test case, which can be executed on this very same screen, is to open up the *Settings* menu. Starting from the currently selected UI element *Channel Search*, this can be achieved by executing the RC commands `[down, right, right, right, right, OK]` in the order they are given, which would visit the UI elements *[Records, Media Center, Screen Share, Tools, Settings]* before opening up the *Settings* menu.

To run the UI test cases, Arcelik uses a setup consisting of a number of hardware components, including a programmable remote controller (Tira [13]) and a connected capture device (UCD [14]). Figure 2 presents a picture of this setup, replicated at Sabanci University. Tira is a programmable remote controller, which is capable of sending any signal that can be sent by a regular remote controller. Being a USB device, it simply replays the pre-recorded signals for the requested RC commands. In this setup, Tira is used to execute the RC commands in the test cases.

The UCD device, on the other hand, is a compact-sized USB device that intercepts the signal sent to the TV panel, such that the video stream can be recreated in a computer. UCD is, indeed, a frequently used device for automated testing of display-related ASICs and display electronics as it offers a flexible and robust method for capturing the images/streams without the need for an external camera, thus avoiding all the associated technical difficulties, including camera calibration. In this setup, Arcelik uses the UCD device to capture the screenshots, so that whether the expected UI states have been reached during and/or after the execution of the test cases, can automatically be determined by analyzing the images. Both devices are connected to a workstation where the drivers for these devices are pre-installed.

In this work, AdapTV uses exactly the setup for model-based test adaptation. More specifically, Tira is used to interact with the TV under test via sending RC signals. UCD is used to capture the screenshots so that the images can be analyzed to determine the current state of the UI. And, the main adaptation logic runs in the workstation.

TABLE 1. Notations used in the paper.

notation	description
S	older version of the system
S'	newer version of the system
M	the model discovered for S
M'	the model discovered for S'
T	the test suite to be adapted
t	a test case in T
P	a path traversed in M
P'	a semantically equivalent path to P in M'
s	a state in M
s'	a state in M'
c	an RC command

III. MOTIVATING EXAMPLE

In this section, we use our running example in Section II to demonstrate, at a smaller scale, the current issues, which Arcelik faces every time the TV user interfaces change. In the aforementioned example, executing the test case `[down, right, right, right, right, OK]` on Grundig TV v02.025.00 starting from the home screen given in Figure 1a, was opening up the *Settings* menu. However, the home screen in the newer version of the TV, namely Grundig TV v04.015.00, has changed as shown in Figure 1b.

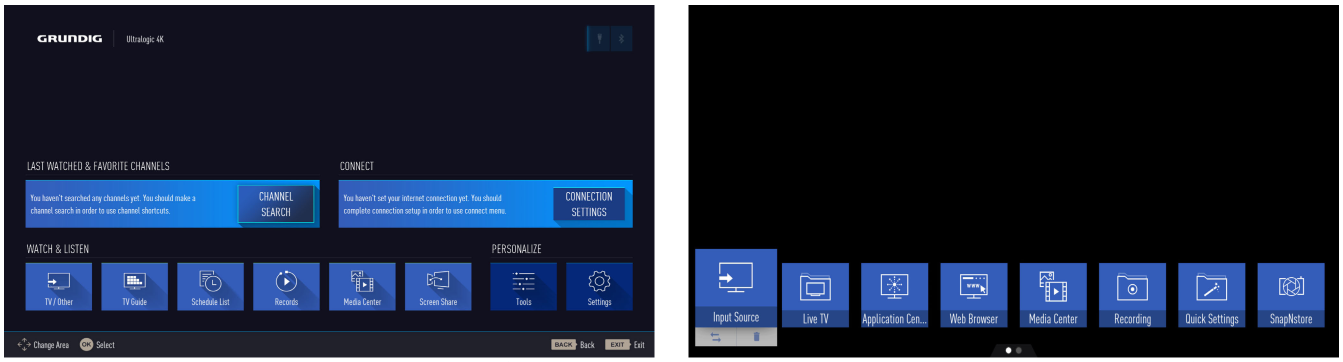
On this newer version, executing the same test case as it is, opens up the *Application Center*, instead of the *Settings* menu. Note, however, that although the UI has changed, the functionality to be tested remains to be intact. Therefore, the same test case is desired to be executed on the newer version. To this end, the test case should be adapted to `[right, right, right, right, right, right, OK]`, which, starting from the currently selected UI element *Input Source* in Figure 1b, opens up the *Quick Settings* menu.

Arcelik has been manually adapting the UI test cases every time the TV user interfaces change. Although many of the commonalities present in the test cases are abstracted away in modules (e.g., in utility functions), it is still a tedious, error-prone, and costly operation for Arcelik to manually fix all the broken test cases. The proposed approach, on the other hand, automatically adapts the test cases in a dynamic and feedback-driven manner.

IV. APPROACH

In this paper, we present a feedback-driven model-based test adaptation approach for smart TVs. From the perspective of the TV software, the proposed approach is a non-intrusive and completely black-box approach, which operates by interpreting the screenshots of the UIs to interact with the TV.

At a very high level, given a test suite T, which is known to be valid for an older version S of the TV, and a new version S' of the TV, to which the test cases are to be adapted, the proposed approach operates in two phases, namely *model discovery phase* and *adaptation phase*. In the model discovery phase, we automatically discover the UI models M and M'



You haven't searched any channels yet. You should make a channel search in order to use channel shortcuts.

You haven't set your internet connection yet. You should complete connection setup in order to use connect menu.

(a) Home screen of Grundig TV v02.025.00. (b) Home screen of Grundig TV v04.015.00.

FIGURE 1. Example screenshots for demonstrating the changes made in the home screens of two different TV versions.

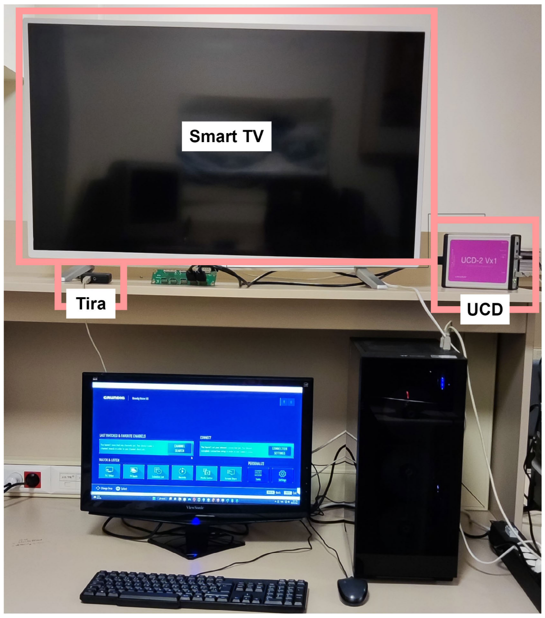


FIGURE 2. Replication of the smart TV testing framework used by Arcelik.

for the older and the new version of the TV, respectively, by opportunistically crawling the UIs. And, in the adaptation phase, each test case $t \in T$ is executed on S and the path P traversed by t in M is determined. The adaptation of t to S' is achieved by first finding a "semantically equivalent path" P' in M' and then dynamically executing this path in a feedback-driven manner on the S'. Table 1 lists some of the notations that we frequently use throughout the paper to represent various aspects of the proposed approach.

Next, we discuss the computational primitives as well as the artifacts used by both the model discovery and the adaptation phases (Sections IV-A-IV-D). Later, we provide the details regarding each phase.

A. UI MODELS

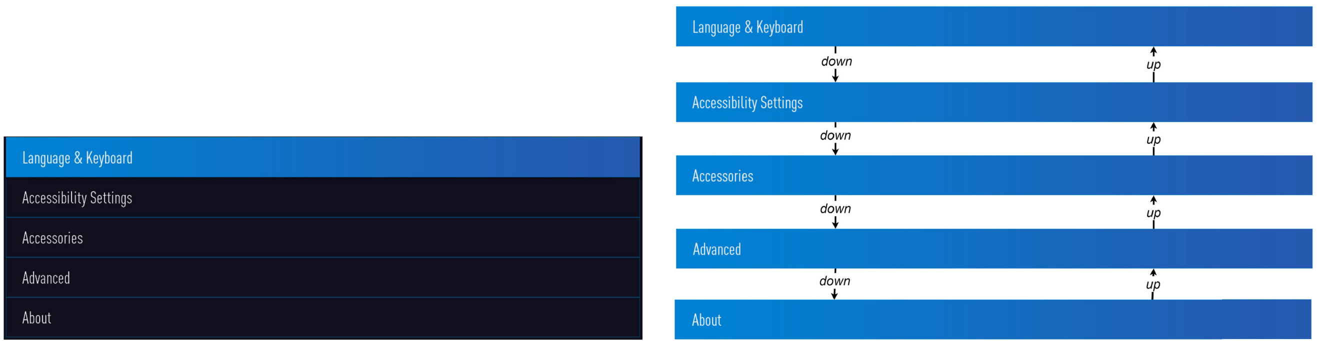
An integral part of the proposed approach is the automatically discovered UI models. The ultimate goal of using a model-based adaptation approach is to minimize the guesswork, which happens when the UI moves to an unexpected state during the adaptation process. Therefore, the UI models we discover should enable us not only to figure out the current state of the UI, but also to get to any state of interest by using the RC commands, so that the adaptation process can be put back on track.

To this end, we express our models as non-deterministic finite state machines. In these models, each state denotes a selection, e.g., a selected UI element, such as a button or an icon. And, the transitions between the states, each of which is labeled with an RC command, represent the possible navigational paths that can be between the UI elements by using the RC.

Note that our models are purposefully non-deterministic because the same RC command on the very same UI element (i.e., on the same model state) may result in a different transition. For example, the behavior of pressing the `back` RC button on a UI element may depend on the actual path followed to get to the element, as this button typically takes the user to the preceding UI element visited. Therefore, depending on the previous state, a different transition can be taken. Although all the scenarios, in which we observed such behavior in our experimental platforms, could have been generalized into a small number of cases, each of which would then be handled in a specialized manner, we decided not to exercise this option and make the models non-deterministic instead, with the goal of improving the generality of the proposed approach.

Figure 3, as an example, presents a simple TV menu encountered in one of our empirical studies together with a UI model automatically discovered for it. Note that, for the sake of simplicity, only a small portion of the actual UI model, which contains two RC commands, namely `up` and `down`, is given in the figure. Furthermore, in this screenshot as well as in the other screenshots presented throughout the paper, we, as needed, use cropped images extracted from the actual screenshots of the entire screens both to make the figures more readable and to save some space.

Going back to our example in Figure 3, the UI model indicates that pressing the `down` RC button when the currently



(a) A simple TV menu. (b) Simplified version of the UI model discovered for the menu in (a).

FIGURE 3. **A simple TV menu (a) and a simplified version of the UI model discovered for it (b).**

selected UI element in the menu is *Language & Keyboard*, would select the UI element *Accessibility Settings*. And, pressing the `down` button again would select *Accessories*, etc. Note that the selected UI element in the menu is highlighted with a bluish background color (Figure 3a). Note further that each state in the UI model (Figure 3b) represents the selected version of the respective UI element. This is because each model state represents the state of the UI after an outgoing transition from the current state is taken, which would select the UI element represented by the target state.

B. DETECTING SELECTIONS

Each state in the UI model corresponds to a selection. Therefore, to map the current state of the UI to a model state, the selection on the screen needs to be determined. To this end, one observation we make is that as smart TVs are meant to be used by a wide spectrum of age groups ranging from children to the elderly, their user interfaces are generally designed, such that the selected elements on the screen can easily be located without any ambiguity. We further observe that one prominent way of achieving this is to box the selected UI element by using a distinctive (and sometimes fading) color either in the form of an outlined box, e.g., with a solid bounding box around the selection, or in the form of a filled box, e.g., with a solid background. For example, in Figure 3 the selected UI element *Language & Keyboard* is highlighted with a distinctive bluish background color.

For this work, we have, therefore, developed a configurable, color-based *detector*. In our context, a detector takes as input a screenshot and returns the cropped image(s) of the selection as output. The way these cropped images are used for mapping the current state of the UI to a state in the UI model will be discussed later in Section IV-C.

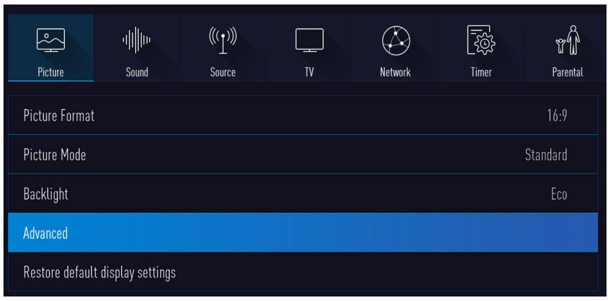
Our detector is configurable in the sense that given an example image of a selection, it figures out the parameters to be used for automatically detecting all the selections of the same sort. Note, however, that the proposed approach can readily be used with any detector as long as the cropped images of the selection are returned.

Given an example image of a selection, we determine the spectrum of colors used in the background and/or in the bounding box of the selected UI element and express it as a filter. Then, given a screenshot, we use the filter to locate the regions of the screen that use the same (or a similar) color spectrum. Next, we apply thresholding to extract the contours [15] as well as the respective bounding boxes and use non-maximum suppression [16] to select a single bounding box from many overlapping bounding boxes. We use an adaptive thresholding algorithm with the Gaussian method for the former. For the latter, we use non-maximum suppression with an overlapping ratio of 30%. Finally, among the remaining bounding boxes, we use additional heuristic-based filters to eliminate the bounding boxes that are highly unlikely to represent an interactable UI element.

The rationale behind these additional filters is to eliminate the bounding boxes that are too small, which may indicate that the respective regions are not meant to be seen by human beings, or that are too large, which may indicate that the respective regions are not meant to be a part of a UI. In the experiments, we, indeed, observed that too small or too large bounding boxes were always caused due to some pixels that could barely be seen (if at all possible) by the naked eye, thus not representing any UI elements.

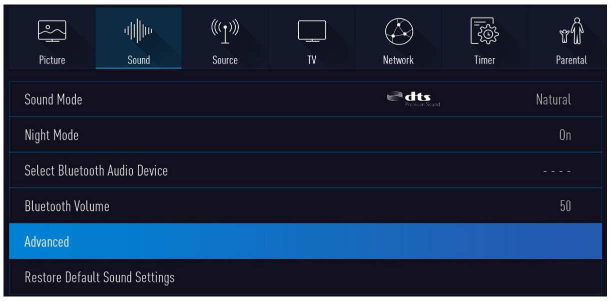
Therefore, we specify the additional filters in terms of the minimum and maximum values for the height, width, and area of the bounding boxes, which are all expressed in relation to the entire screenshot. Note that these filters can also be used to detect the selected UI elements based on their sizes. This is important because an alternative way of visualizing the selected elements is to make them larger than the other comparable elements on the screen.

One observation we make is that the TV UIs also use visual clues (as needed) to indicate the contexts for the selected UI elements. It turns out that determining the context of a selection is an important task because the same UI element may be present in different contexts with different functionalities. For example, in Figure 4a, while the current selection is *Advanced*, the context for this selection is *Picture*. There is, however, another UI element with exactly the same label



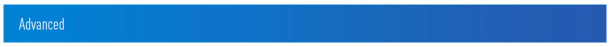
Picture Format 16:9
Picture Mode Standard
Backlight Eco
Advanced
Restore default display settings

(a) The *Picture* menu

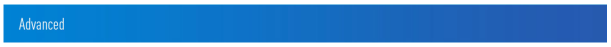


Sound Mode Natural
Night Mode On
Select Bluetooth Audio Device - - - -
Bluetooth Volume 50
Advanced
Restore Default Sound Settings

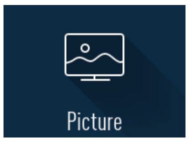
(b) The *Sound* menu



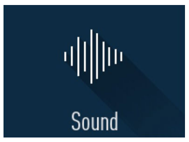
(c) Selected UI element in (a)



(d) Selected UI element in (b)



(e) The context for the selection in (a)



(f) The context for the selection in (b)

FIGURE 4. Detecting contexts for the selections.

and with exactly the same look and feel, but in the context of *Sound* (Figure 4b). While the former opens up the advanced settings menu for the pictures, the latter does that for the sound. As the functionalities of these identical-looking UI elements are different, they should map to different states in the UI model.

In this work, we use our detector logic to also determine the contexts. More specifically, a *primary detector*, which is used for detecting the selected UI elements, can be associated with many (zero or more) *context detectors*, which are used for detecting the contexts for the selections. Going back to our example in Figure 4a, while the primary detector would return the cropped image in Figure 4c as the selected UI element, the associated context detector would return the cropped image in Figure 4e as the context for the selection.

Another observation we make is that there may be multiple different ways of visualizing the selected UI elements. For example, the way the selections are visualized in the settings menu may be different than the way they are visualized on the home screen. To account for this, we allow multiple primary detectors to be defined, each of which may come with its own associated set of context detectors. In the presence of multiple primary detectors, each detector is activated separately and the cropped images obtained from the detectors are collected for later processing (Section IV-C). Note, however, that alternative ways of visualizing the selections are typically employed for different parts of the TV. That is, on a single screen there is typically only one way of visualizing the selected UI elements, in which case only one primary detector would return some cropped images. This was, indeed, the case in our experiments where we never had multiple primary detectors returning cropped images for a single screen.

C. *WhereAmI: MAPPING SELECTIONS TO MODEL STATES*

At a very high level, a detector returns a (possibly empty) collection of rectangular images cropped from the screenshot, collectively representing the selection on the screen. Given such a collection, we merge the images into a single image, called the *ID-image*, such that the resulting image can be used to map the selection to a state in the UI model. Note that for a primary detector, which is associated with a set of context detectors, the ID-image is formed for the collection of all the images returned from the detectors.

To form the ID-image for a given collection of images, we first sort the images by the coordinates of their upper-left corners and then append them one after another. Note that the sorting operation guarantees that, given the same collection of images, the same ID-image can always be deterministically formed.

After computing the ID-image for a selection, the next step is to map the ID-image to a state in the UI model. We refer to this functionality as the *WhereAmI* functionality. In the model discovery phase, WhereAmI is used to figure out whether the current UI state has been encountered so far or not, which determines if the underlying UI model needs to be populated with a new state. In the adaptation phase, WhereAmI is used to figure out where the TV is currently at in the UI model after executing an RC command, which determines the next RC command to be executed to reach a target state.

To map the UI states to the model states, we associate a model state with the ID-image of the selection represented by the state. Therefore, one way of mapping a selection to a model state is to compare the respective ID-images. This, however, turns out to be a costly solution due to its space and runtime overheads. The space overhead of the aforementioned solution is high because all the ID-images may need to be loaded into the memory for the comparisons. The runtime overhead is also high because the ID-image of the selection may need to be compared to all the ID-images associated with the model states and comparing images is costly. Note further that these overheads should be paid at every iteration of both the model discovery and the adaptation phases as mapping the current UI state to a model state needs to be performed after executing every RC command.

We, therefore, use a hashing function, called Locality Sensitive Hashing (LSH), which maps similar images to the close-by hash values [17]. Consequently, rather than keeping the actual ID-images for the model states, we keep their hash values. We, in particular, use the dHash algorithm, which basically operates by computing the differences between the adjacent pairs of pixels [18].

Given an ID-image, we first resize it to a 128x129 pixel (i.e., 128 rows by 129 columns) image by using bilinear interpolation. Then, we convert the image to grayscale as grayscale images tend to produce better results. Next, for each row in the resized image starting from the first row, we iterate over all the pixels in the row from the right-most to the left-most pixel. For each adjacent pair of pixels, we keep one bit of information, the value of which is 1 if the first pixel value in the pair is smaller than the second one and 0 otherwise. Consequently, for a given resized ID-image, we compute a 16384-bit ($128 * 128$-bit) hash value.

The number of bits for storing the hash values, thus the dimensions of the resized ID-images, is selected, such that the subtle, but relevant differences between the ID-images, thus the selections, can be detected. To this end, the UI elements, which slightly change their appearances depending on the RC commands executed on them, can be used. The *Bluetooth Volume* menu item given in Figure 4b is an example of such an element. As the volume is increased, the number on the right-hand side of the element increases by one. So, the smallest number of bits that can distinguish between different Bluetooth volumes, such as between volume 0 and volume 1, is a good candidate to use, which, indeed, was exactly what we did to determine the size of the hash values (i.e., 16384 bits) in the experiments.

To compare two hash values for determining whether the respective ID-images, thus the selections, are the same, we use Hamming distance by taking the potential noise in the images into account. More specifically, we compute the number of bits that are different between the bit representations of the hash values. The smaller the distance, the more the respective ID-images, thus the respective selections, tend to be similar.

Therefore, to map a selection to a model state, we first compute the hash of the ID-image constructed for the selection and then compare it to the hash values associated with the previously discovered model states. If the minimum difference between the hash value of the selection and that of a model state is smaller than a predetermined threshold (in our case, 80), the selection is mapped to the state. If no such state can be found, then the selection is considered to not have been encountered before. In the presence of multiple potential equivalent states, the state with the minimum distance is selected. The aforementioned threshold value is selected, such that the effect of the noise in the ID-images is reduced as much as possible, yet the relevant differences between the selections can still be detected.

D. TEST CASES

In this work, the TV under test is treated as a black box. We, therefore, express a test case as a sequence of RC commands that can be carried out against the TV. That is, a test case in our context corresponds to a path in a UI model (Section IV-A).

E. MODEL DISCOVERY PHASE

We automatically discover the UI models for both the older and the new version of the TV by opportunistically crawling the user interfaces. At a very high level, given a set of RC commands to be used for the interactions, the crawler aims to execute each command on every interactable UI element in all the screens at least once.

Algorithm 1 describes the model discovery process. At each iteration of the process, after executing an RC command, the screenshot of the UI is taken (line 10) with the help of the UCD device (Section II). Then, the screenshot is analyzed to determine the selection (line 11, Section IV-B). If a state representing the selection is not present in the UI model that has been discovered so far, the model is populated with the newly discovered state (line 12). In either case, the state representing the selection is marked as the current state (line 14). While doing so, the transition labeled with the RC command executed, which took the system from the previous state to the current state, is inserted into the model, if not already present in the model (line 13). Next, the RC command to be executed in the subsequent iteration is determined in an opportunistic manner by steering the crawler to the less-explored parts of the UI model, so that each RC command has a chance of being executed on every state (line 8).

Algorithm 2 presents the details of the *next_cmd* function in Algorithm 1 (line 8). The crawler first figures out whether there is any RC command, which has not yet been executed on the current state (line 2). If so, one RC command among all such commands is chosen randomly (line 4) while giving precedence to the navigational commands (i.e., the arrow buttons on the RC) with the goal of discovering more states as early as possible. Otherwise, i.e., when each RC command has been executed at least once on the current state, then the nearest state with some missing outgoing transitions (i.e., with some RC commands that have not yet been executed on the state) is determined (line 6) by using a breadth-first search (BFS) traversal of the UI model starting from the current state (i.e., by using an unweighted shortest path algorithm).

Algorithm 1 Model Discovery

```
1:  function discoverModel(Model $M$)
2:      Reboot the TV, if needed
3:      $screen \leftarrow take\_screenshot()$
4:      $s_{prev} \leftarrow whereAmI(M, screen)$
5:      Mark $s_{prev}$ as global entry state
6:      Populate $M$ with $s_{prev}$, if needed
7:      while the termination criterion has not been reached do
8:          $cmd \leftarrow next\_cmd(M, s_{prev})$
9:          $execute\_cmd(cmd)$
10:         $screen \leftarrow take\_screenshot()$
11:         $s_{curr} \leftarrow whereAmI(screen)$
12:         Populate $M$ with $s_{curr}$, if needed
13:         Populate $M$ with the transition from $s_{prev}$ to $s_{curr}$ labeled $cmd$, if needed
14:         $s_{prev} \leftarrow s_{curr}$
15:     end while
16:     return $M$
17: end function
```

Algorithm 2 Opportunistic Crawling

```
1:  function next_cmd(Model $M$, State $s$)
2:      Let $cmds_{missing}$ be the missing RC commands on $s$
3:      if $cmds_{missing} \neq \emptyset$ then
4:          Let $cmd_{next}$ be a randomly selected RC command in $cmds_{missing}$
5:      else
6:          Let $s_{target}$ be the nearest state from $s$ with some missing RC commands in $M$
7:          if $s_{target} == None$ then
8:              Raise the termination signal for the discovery process
9:              return
10:         end if
11:         Let $shortest\_path$ be the shortest path from $s$ to $s_{target}$ in $M$
12:         Let $cmd_{next}$ be the first RC command to be executed in $shortest\_path$
13:     end if
14:     return $cmd_{next}$
15: end function
```

In the BFS traversal, the shortest path to the nearest state selected is also computed (line 11). Then, the crawling process is opportunistically steered towards the selected state by choosing the first RC command in the computed shortest path as the RC command to be executed in the subsequent iteration (line 12).

Note that we don't force the search to reach the selected state before the discovery process can resume. More specifically, if everything goes as predicted by the underlying UI model, the selected state will be reached and a missing RC command on this state will be executed. If, however, on the path to the selected state, a state with some missing RC commands was encountered (such as a state unexpectedly visited due to the non-determinism in the TV), the discovery process would resume with executing a randomly chosen missing command on the state (line 9 in Algorithm 1), thus the initial goal of reaching the nearest state selected, would gracefully be discarded.

Note that, as indicated by Algorithm 1, the model discovery process always starts with rebooting the TV (line 2). This is mainly for detecting the *global entry states* in the UI models (line 3-5), which are, indeed, the first states that are discovered right after the TV is rebooted. We reboot the TV before starting the crawling because otherwise, the global entry states would depend on the state of the TV, from which the crawling has started. Furthermore, the model discovery process (Algorithm 1) takes as input the current version of the UI model (line 1). Therefore, the process can be started either with an empty model or with a pre-populated model without requiring any additional operations as the current state of the UI is determined before starting the discovery process (lines 3-5). This enables the model discovery process to be stopped and started as needed.

The iterations of the discovery process end when for each distinct pair of model state and RC command, there is at least one outgoing transition from the state labeled with the RC command. This primary stopping criterion is, indeed, further augmented with an optional secondary stopping criterion, which terminates the discovery process after a predetermined number of RC commands have been executed. Having a secondary stopping criterion is practical, especially for preventing potential infinite loops and/or for taking the amount of available resources for the discovery process into account.

F. ADAPTATION PHASE

Given a test suite T to be adapted and the UI models M and M' that are automatically discovered for the older version S and the new version S' of the TV, respectively, the next step is to adapt each test case $t \in T$ from S to S'. To this end, we first execute t on S and determine the path P taken by it in M, which is indeed a sequence of states (Section IV-F3).

The problem of adapting t to S' is then cast to the problem of finding a semantically equivalent path P' in M'. Note that there are multiple ways of defining the equivalency relation between the paths. For this work, we, in particular, distinguish between two different equivalency criteria: *route equivalency* and *destination equivalency* (Section IV-F2). While the former seeks for visiting an equivalent state in M' for each state in P, the latter aims to visit an equivalent state only for the destination state (i.e., the last state) in P, regardless of the intermediate states visited. Note that both criteria depend on the notion of *state equivalency*.

1) STATE EQUIVALENCY

To determine whether a state s in M is equivalent to a state s' in M', we use a weighted similarity measure consisting of three metrics, namely *semantic similarity*, *lexical similarity*, and *likelihood*. Given s and s', the first two metrics leverage the information conveyed in the ID-images associated with s and s' and they do this regardless of the current adaptation process. That is, given the same states, these metrics always return the same value. The last metric, however, takes the current adaptation process into account by favoring the states that are closer to the one being currently visited by the adaptation process.

Regarding the first two metrics, one observation we make is that the UI elements in smart TVs are typically associated with explanatory texts, such as labels. Even when an image-only icon appears on a screen, it is often the case that either the icon has some embedded human-readable text or the icon gets a label as soon as it is selected. We believe that this is because the user interfaces of the TVs are designed in such a way that allows them to be easily used even for the first time by a wide spectrum of age groups without any training.

Without losing the generality of the proposed approach, we, therefore, opted to compute the semantic and lexical similarities based on the texts extracted from the ID-images. Note that directly comparing the ID-images, such as by using their hash values as we did in the model discovery phase (Section IV-E), is not a viable solution because, in the adaptation phase, we are interested in comparing the selections that come from two different versions of the TV. Therefore, the look and feel of the UI elements, such as the icons, may differ between the versions. Furthermore, in the absence of any text associated with a selection, such as the presence of an image-only icon, image classification techniques [19] can be used to label the icon first. Then, the label (possibly together with the explanatory text that comes with it) can be used with the semantic and lexical similarity metrics proposed in this work. The accuracy of the classifications can even further be improved by training domain-specific classifiers as the user interfaces in the TV domain tend to use the same or similar icons for the same or similar functionalities. We, however, leave this part as future work.

a: EXTRACTING TEXTS

To extract the texts from the ID-images, we use optical character recognition (OCR) [20]. We do this in two stages: *text detection* and *text recognition*. In the text detection stage, we determine the boundaries of the regions in the given image [21], which are likely to contain some texts. To this end, we use a pre-trained model, called *craft_mlt_25k*, which was trained by using the ICDAR datasets with 25 thousand iterations for fine tuning [22]. Furthermore, we utilize non-maximum suppression [16] for merging the overlapping regions.

After the potential text regions in the ID-image are identified, we, in the text recognition stage, apply OCR to the cropped regions only, rather than to the entire image. We do this because our feasibility studies demonstrate that the former generally performs better than the latter in terms of OCR accuracy. In the presence of multiple text regions, the cropped images are first ordered by their upper-left corners and then the text is extracted. For text recognition, we use a pre-trained model, called *TPS-ResNet-BiLSTM-Attn-case-sensitive*, which was trained by using a large number of synthetic and real-world datasets [23].

Once the text from an ID-image is extracted, it is associated with the respective state in the UI model, so that the extraction task is not repeated redundantly. That is, for each state in a UI model, the text is extracted only once.

b: SEMANTIC SIMILARITY

Given two states, the semantic similarity between them is computed as a measure of how close the meanings of the texts associated with the states are. To this end, we developed a similarity metric inspired by Rau et al. [24].

Given two states s and s' where $s.text$ and $s'.text$ represent the texts extracted from these states, respectively, we first preprocess each text by tokenizing it, removing the non-literal characters, and applying stemming. We, however, choose not to eliminate the stop words as they can play an important role in our context, such as helping to identify the difference between "sign in" and "sign up."

$$
\begin{aligned}
\texttt{sem\_sim("TV Shows","Music Videos")} &= \left[\begin{array}{ccc} & \texttt{music} & \texttt{video} \\ \texttt{tv} & \mathbf{0.24} & \mathbf{0.32} \\ \texttt{show} & 0.21 & 0.22 \end{array}\right] \\
&= \frac{0.24 + 0.32}{2} = 0.28
\end{aligned}
$$

FIGURE 5. Semantic similarity computation between state s and s' where $s.text$ ="TV Shows" and $s'.text$ = "Music Videos."

We then form an nxm matrix where n and m are the numbers of extracted tokens in $s.text$ and $s'.text$, respectively. The value stored in each cell of this matrix represents the cosine similarity between the respective tokens. To this end, we use Google's pre-trained word2vec model, which has a vocabulary of about 3 million words and phrases trained by using about 100 billion words from a Google News dataset [25].

Finally, the semantic similarity between s and s' is computed as the mean of the maximum cosine similarities reported in the columns of the constructed matrix. To deterministically compute the similarities (which can be an issue when $n \neq m$), we construct the matrix, such that $n \leq m$ always holds true. Note that the semantic similarity metric assumes a value between 0 and 1, inclusive. The higher the value, the more the texts are semantically similar to each other.

Figure 5 presents an example where the semantic similarity between two states with the extracted texts "TV Shows" and "Music Videos" is computed. After pre-processing, we have two bags of tokens: {tv, show} and {music, video}. Thus, a 2×2 matrix is formed and the pairwise similarities between the tokens are computed by using Google's word2vec model. Then, the maximum value in each column (0.24 and 0.32) is determined and their arithmetic average (0.28) is computed as the semantic similarity score between the texts.

c: LEXICAL SIMILARITY

While the semantic similarity metric aims to quantify the similarity in meanings, the lexical similarity metric quantifies the lexical similarity by using a normalized edit distance function.

One reason as to why we need an additional similarity metric is that not all the tokens extracted from the texts may be included in the vocabulary of the word embedding model being used. For example, the word "Grundig," which fairly frequently appeared on the TV menus in the experiments as it is another brand name owned by Arcelik, was not a part of Google's word2vec model. Since nothing would be known about these tokens, no semantic similarity could be computed. In these cases, we use lexical similarity instead.

Another reason is that when one set of tokens were subsumed by another set, the semantic similarity computation discussed above always produces a perfect score, which could be misleading. For example, if one of the texts contains only one token, namely "TV," then this text will have a perfect semantic similarity score with any texts containing the same word, including "TV," "TV Shows," "TV Movies," and "TV Games," where one would expect to have a higher score for the first choice as it represents an exact match. Therefore, in these cases, we not only account for the semantic similarity but also take the lexical similarity into account.

Given two states s and s', we compute the lexical similarity by using the following normalized edit distance formula:

$$lex\_sim(s, s') = 1 - \frac{edit\_distance(s.text, s', text)}{max(|s.text|, |s'.text|)} \tag{1}$$

where $|*.text|$ operator returns the number of characters in the given text and the $edit\_distance$ function computes the edit distance between two texts, i.e., the minimum number of operations required to transfer one text into the other.

d: LIKELIHOOD

Given the current UI state s', the likelihood for state s'' to be the next target state to visit during an adaptation process, drops exponentially with the distance between the states:

$$likelihood(s', s'') = e^{-d/20} \tag{2}$$

where d is the length of the shortest path from s' to s'' in the underlying UI model.

As discussed in Section IV-B, there may be multiple UI elements with the same look and feel, e.g., with the same textual information, appearing in different parts of the TV with different functionalities. For example, a UI element with the label Music may appear both on the home screen and in the settings menu, serving different purposes. While the former enables the user to play music, the latter allows changing the music-related settings. If we were to use only the semantic and lexical similarity metrics, both of these UI elements would get the same similarity score. The likelihood metric, however, favors the UI element that can be reached by executing fewer number of RC commands. That is, it favors the states that are closer to the current state, rather than the states that are farther away. The rationale behind this metric is that, in a sequence of RC commands, the preceding commands typically determine the context for the current command. For example, if a test case aims to evaluate whether the music-related settings can be configured correctly or not, the test case needs to visit the settings menu first. Therefore, if the TV is already in the settings menu and we are seeking to interact with a UI element labeled *Music*, it is highly unlikely that the element we are looking for is the one on the home screen. After all, if the *Music* item on the home screen was meant to be visited, the original test case would have executed additional commands to change the context accordingly by visiting the home screen.

e: WEIGHTED STATE EQUIVALENCY SCORE

Given the current state s' in M' and the target state s in M, the equivalent state of which needs to be visited in M', we compute the state equivalency score of a state s'' in M' by computing the weighted sum of the scores obtained from the previous three metrics:

$$\begin{aligned} state\_eq\_scr(s, s', s'') = {} & w_1 * sem\_sim(s, s'') \\ & + w_2 * lex\_sim(s, s'') \\ & + w_3 * likelihood(s', s'') \end{aligned} \tag{3}$$

If the semantic similarity score can be computed (i.e., if all the tokens are in the vocabulary of the word2vec model), we use $w_1 = 0.6$, $w_2 = 0.2$, and $w_3 = 0.2$. We assign more weight to the semantic similarity because, between different versions of the TV, even when the textual information associated with a UI element gets changed, the meaning is typically preserved (e.g., changing the "Next" button to a "Proceed" button). If, however, the semantic similarity cannot be computed, we use $w_1 = 0$, $w_2 = 0.8$, and $w_3 = 0.2$. We value lexical similarity more in these cases because we first look for some sort of similarity before taking the physical proximity into account.

f: FINDING THE EQUIVALENT STATES

Given the current state s' in M' and the target state s in M, the equivalent state $\tilde{s}$ of s in M' is determined by first computing the weighted equivalence score between s and every state in M', i.e., by calling $state\_eq\_scr(s, s', s'')$ for each s'' in M'. The scores are then sorted in reverse order and the top p states (in our case, $p = 1$) are chosen as the candidate equivalent states of s. This functionality will be referred to as *find_equivalent_state* in the remainder of the paper.

2) PATH EQUIVALENCY

Given the notion of state equivalency, we define two different types of path equivalency criteria, which are inspired by the test oracles used for user interface testing in the TV domain, namely *route equivalency* and *destination equivalency*.

The route equivalency aims to visit an equivalent state in M' for every state in P while preserving the order, in which the states are visited:

Definition 1: Given the path $P = [s_0, s_1, \dots, s_k]$, which is a sequence of states traversed by a test case t in M during its execution on S, the *route equivalency* criterion is achieved when t is adapted to a test case, which makes S' to follow a path $P' = [s'_0, s'_1, \dots, s'_l]$ in M' where for each state s_i in P $(0 \leq i \leq k)$, an equivalent state s'_{l_i} exists in P', such that $l_0 < l_1 < \cdots < l_k$.

Note that due to the changes in the user interfaces, additional states may need to be visited in M' to achieve the route equivalency. That is, the equivalent states of interest in M' do not need to be visited in a consecutive manner.

Destination equivalency, on the other hand, aims to visit an equivalent state of just the destination state (i.e., the last state) in P regardless of the path taken (i.e., the intermediate states visited).

Definition 2: Given a path $P = [s_0, s_1, \dots, s_k]$ traversed by test case t in M during its execution on S, the *destination equivalency* criterion is achieved when t is adapted to a test case, which makes S' to follow a path $P' = [s'_0, s'_1, \dots, s'_l]$ in M', such that s'_l is an equivalent state of s_k.

3) TEST ADAPTATION

Algorithm 3 presents the details of adapting a test case t from S to S' given the automatically discovered models M and M'. The aforementioned algorithm, indeed, aims to achieve route equivalency. After discussing this algorithm in detail, we provide details regarding how the algorithm can be adapted to achieve destination equivalency.

a: RUNNING THE TEST CASE ON THE OLD VERSION

The first step of the adaptation process is to run t on S and determine the path P taken by t in M (lines 2-9). To this end, the TV running version S is rebooted, if needed (line 2). Rebooting the TV is generally required because the TV under test typically needs to be rebooted in between the test runs to make sure that the order, in which the test cases are executed, does not affect the test results. Note, however, that the proposed approach does not actually depend on rebooting the TV in between the test runs. For example, the test cases may come with their own tear-down sequences or no tear-down sequence may actually be needed. The AdapTV will be agnostic to that since it always starts with figuring out the current state of the UI before executing a test case (line 7).

We then execute the RC commands in t, which is indeed expressed as a sequence of RC commands $[c_1, c_2, \dots, c_k]$ (line 5). After executing an RC command c_i $(1 \leq i \leq k)$, we use the WhereAmI functionality (Section IV-C) to determine the resulting state s_i of the system in M (line 7). The result is a path $P = [s_0, s_1, \dots, s_k]$ traversed by t in M. Note that, throughout the paper, the states subscripted with 0, such as s_0 and s'_0, are used to refer to the global entry states. Note further that the proposed approach keeps on updating the underlying UI model even in the adaptation phase. For example, if there is no previously known transition from s_{i-1} to s_i with the label c_i in M, then M is populated with the newly discovered transition and with the state s_i (if needed).

b: FINDING THE EQUIVALENT STATES

Given the path $P = [s_0, s_1, \dots, s_k]$ in M, the next step is to find the respective equivalent states in M' (lines 10-16).

Algorithm 3 Adaptation (Route Equivalency)

```
1:  function adapt($Test\ t$, $Model\ M$, $Model\ M'$)
2:      Reboot the TV running the older version, if needed
3:      $P = []$
4:      for each $cmd$ in $t$ do
5:          $execute\_cmd(cmd)$
6:          $screen \leftarrow take\_screenshot()$
7:          $s \leftarrow whereAmI(screen)$
8:          $P \leftarrow append(P, s)$
9:      end for

10:     $\tilde{P} = []$
11:     $\tilde{s}_{prev} \leftarrow None$
12:     for each $s$ in $P$ do
13:         $\tilde{s}_{curr} \leftarrow find\_equivalent\_state(s, \tilde{s}_{prev})$
14:         $\tilde{P} \leftarrow append(\tilde{P}, \tilde{s}_{curr})$
15:         $\tilde{s}_{prev} \leftarrow s_{curr}$
16:     end for

17:     $t' \leftarrow []$
18:     Reboot the TV running the new version, if needed
19:     $screen \leftarrow take\_screenshot()$
20:     $s'_{curr} \leftarrow whereAmI(M', screen)$
21:     for each $\tilde{s}$ in $\tilde{P}$ do
22:         Clear all the "deleted" labels in $M'$
23:         $equivalent\_state\_visited \leftarrow false$
24:         $trial\_cnt \leftarrow 0$
25:         while (not $equivalent\_state\_visited$) and ($trial\_cnt < MAX\_ALLOWED$) do
26:             Let $shortest\_path$ be the shortest path from $s'_{curr}$ to $\tilde{s}$ in $M'$
27:             Let $cmd$ be the first RC command to be executed in $shortest\_path$
28:             Let $s'_{expected}$ be the expected state to be visited after executing $cmd$ according to $M'$
29:             $t' \leftarrow append(t', cmd)$
30:             $execute\_cmd(cmd)$
31:             $screen \leftarrow take\_screenshot()$
32:             $s'_{prev} \leftarrow s'_{curr}$
33:             $s'_{curr} \leftarrow whereAmI(M', screen)$
34:             if $s'_{curr} \neq s'_{expected}$ then
35:                 Mark the transition from $s'_{prev}$ to $s'_{expected}$ with the label $cmd$ in $M'$as deleted
36:             end if
37:             if $s'_{curr} == \tilde{s}$ then
38:                 $equivalent\_state\_visited \leftarrow true$
39:             end if
40:             $trial\_cnt \leftarrow trial\_cnt + 1$
41:         end while
42:         if  not $equivalent\_state\_visited$ then
43:             Mark $\tilde{s}$ as unvisited
44:         end if
45:     end for
46:     return $t'$
47: end function
```

To this end, for each state s_i in P, we find the equivalent state $\tilde{s}_i$ in M' (line 13) and populate the list of equivalent states $\tilde{P}$ accordingly (line 14).

Note that the *find_equivalent_state* function (line 13) can return an ordered list of multiple candidate states (i.e., the top p candidates) for the current state of interest (Section IV-F1).

Therefore, the search space for the adaptation process can, indeed, be represented as a tree, where each node represents a state s to be visited in M and its outgoing edges represent the candidate list of equivalent states in M' sorted in the reverse order by their weighted equivalency scores. Then, the adaptation process is carried out in a depth-first fashion where each path from the root node to a leaf node in the tree represents a candidate sequence of equivalent states. This enables the process to run in a backtracking manner. That is if an equivalent state of interest cannot be visited in M', the process can backtrack and exercise an alternative candidate state. The details of this mechanism, which should be clear by now, are not given in Algorithm 3 for a better presentation of the algorithm. After all, we, in this work, worked with only the top-scored (i.e., $p = 1$) candidate states.

c: ADAPTING THE TEST CASE

Given a sequence of equivalent states $\tilde{P} = [\tilde{s}_0, \tilde{s}_1, \ldots, \tilde{s}_k]$, the adaptation is carried out in a feedback-driven manner by executing RC commands on the new version S' of the TV, such that the equivalent states of interest in $\tilde{P}$ are visited in the underlying UI model M'. To this end, we first reboot the TV running version S' (if needed) and determine the initial state s'_0 by using the WhereAmI functionality (line 20).

Then, for each state, $\tilde{s}$ in $\tilde{P}$, the minimum sequence of RC commands, which would take the TV from its current state s'_{curr} to the state $\tilde{s}$, is determined by performing a BFS in M' starting from s'_{curr}, i.e., by using an unweighted shortest path algorithm (line 26). Rather than executing all the RC commands on the computed path to get to $\tilde{s}$, we do it in a feedback-driven manner (lines 30-36). That is, after executing the very first command *cmd* on the path (line 30), which would move the TV towards $\tilde{s}$, we determine the resulting UI state (lines 31-33). We do this because, in the presence of non-determinism (Section IV-E), the TV, after executing *cmd*, might have moved to a different state than the one predicted by the UI model. Therefore, in the subsequent iteration, a new sequence of RC commands is computed (line 26) starting from the actual current state of the UI, rather than sticking to the original plan. The iterations end when the target state $\tilde{s}$ has been reached (line 38) or a maximum number of allowed RC commands have been executed without any success, in which case the state is marked as unvisited (line 43) and the adaptation process resumes with the subsequent state in $\tilde{P}$.

One observation we make is that the non-deterministic transitions, i.e., when the system ends up in a different state than the one predicted by the UI model (line 34), are typically caused due to the transitions between different menus. When, for example, there are two menus, between which one can go back and forth by using the `right` and `left` RC buttons, the UI element which would be selected after using one of these buttons, depends on the last UI element that was selected in the target menu, regardless of the UI element selected in the current menu, or vice versa.

One way the aforementioned non-determinism might affect the test adaptation process is that the process could get stuck with a non-deterministic path, which may not be executed as predicted by the underlying UI model, starving the adaptation process. To overcome this issue, when a transition takes the TV to a different state than the one predicted by the model, we temporarily marked the transition as deleted (line 35), preventing the transition to be taken again. Note that this strategy is well aligned with our observation that non-deterministic transitions typically occur between different menus. More specifically, for such a transition, the source state and the target state are located in different menus. Even if taking such a transition may not take the system to the expected UI element specified in the model, it would make the TV move the menu, in which the expected UI element is located. Since within the same menu, we typically don't observe any non-deterministic behavior, the expected UI element can be reached in a deterministic manner, even when the respective non-deterministic transition taken between the two menus is marked as deleted. After all, the transitions marked as deleted will be cleared after the current target state $\tilde{s}$ has been visited (line 22).

We have so far discussed the adaptation process for achieving route equivalency, achieving destination equivalency is similar except for line 21. More specifically, rather than feeding the search process with the entire sequence of equivalent states $\tilde{P} = [\tilde{s}_0, \tilde{s}_1, \ldots, \tilde{s}_k]$, we feed it with only the last state $\tilde{P} = [\tilde{s}_k]$. The rest of the algorithm operates exactly in the same manner.

Note that, regardless of whether the route equivalency or the destination equivalency criterion is being used, the way we determine the sequence of equivalent states $\tilde{P}$ stays the same (lines 10-16). That is, even for the destination equivalency, although only the last state in $\tilde{P}$ will be used as the target state to be reached, we determine the equivalent states for all the preceding states one after another before the equivalent state of the last state can be determined. We do this to capture the contexts in P so that the equivalent states (especially the one for the destination state) can reliably be determined. Note that, in a given iteration, the state returned from the *find_equivalent_state* function (line 13) depends on where the TV is currently at in M' after visiting all the preceding states in P. This is mainly due to the likelihood metric used in the equivalency score computations (Section IV-F1). Therefore, when we are determining the equivalent state for the destination state we take the context (i.e., how the destination state was reached in the original test case) into account. Otherwise, among all the possible UI elements that have the same look and feel as the destination state, always the one, which is closer to a global entry state would be selected.

V. EXPERIMENTS

To evaluate the proposed approach, we have carried out a series of experiments.

A. SUBJECT TVs

In these experiments, we used two different versions of a Grundig TV as the subject TVs, namely v02.025.00 and v04.015.00. We, in particular, picked these versions for the study because the changes made between them were typical of the UI changes made in TVs by Arcelik. Figures 1a and 1b present the home screen of the older and the new version, respectively, demonstrating the changes made in this single screen.

B. SUBJECT TEST SUITES

As a test suite to be adapted, we started with around 2000 in test cases, which have been developed and used by Arcelik for testing the user interfaces of their latest TVs, encoding the decades of experience Arcelik has gained in testing TVs. To reduce the potential biases in the analysis, we actually utilized a subset of these test cases. More specifically, one observation we make is that changes in the user interfaces of the TVs tend to affect the higher-level menus, whereas the lower-level menus mostly stay intact. This is mainly because the lower-level menus typically correspond to the most fundamental functionalities of the TV, which are less likely to get changed. Therefore, to avoid any biases in the analysis, among all the test cases that arrive at the same menu, which did not get changed between the versions, we randomly picked a single test case and used it in the experiments. The rationale was simple; once such a menu has been reached during the adaptation, visiting any UI element in that menu was not an issue as the menu did not get changed between the versions. Therefore, including all the test cases ending in such a menu can introduce a bias in the analysis as the number of related test cases may depend on the number of available UI elements in the menu. All told, we culled 60 real test cases from the Arcelik test suite.

On top of these real test cases, to achieve 100% coverage of all the menus in the TV, we also generated some random test cases, such that each menu, which is not exercised by the real test cases, is exercised by at least one test case. To this end, we generated and experimented with 60 additional test cases, each of which had between 1 and 23 RC commands, inclusive. These test cases also helped us evaluate the sensitivity of the proposed approach to the test case lengths.

C. EVALUATION FRAMEWORK

We evaluate both the effectiveness and the efficiency of the proposed approach. More specifically, we measure the effectiveness of an adaptation strategy in terms of its *success rate* and *adaptation rate*, whereas the efficiency is measured in terms of the *test length overhead* and *runtime overhead*.

1) SUCCESS RATE

We indicate the success of an adaptation process as a boolean value indicating whether the test case is successfully adapted or not. Note that we evaluate the success always with respect to the ground truth, i.e., the sequence of *true* equivalent states that need to be visited for a given test case. To identify the ground truths in the experiments, we first studied the TVs and carried out small-scale experiments as needed. We then validated all the ground truths with the respective testing team at Arcelik.

Definition 3: For the route equivalency criterion, a test case is said to be *successfully adapted* (or the adaptation process for the test case is considered to be *successful*), if the route equivalency criterion (Definition 1) is achieved with respect to the ground truth.

Definition 4: For the destination equivalency criterion, a test case is said to be *successfully adapted* (or the adaptation process for the test case is considered to be *successful*), if the destination equivalency criterion (Definition 2) is achieved with respect to the ground truth.

We then define the success rate as follows:

Definition 5: Given a collection of test cases to be adapted, *success rate* is the percentage of the successfully adapted test cases.

2) ADAPTATION RATE

While the success rate quantifies the success of a test adaptation process as a phenomenon with a binary result, i.e., whether the test case has been successfully adapted or not, the adaptation rate aims to measure the ratio of the successfully visited states.

Definition 6: In an iteration of the adaptation process where the goal is to visit a true equivalent state $\hat{s}$ of a state s, which is visited in the older version of the TV, s is said to be *successfully visited* (or, for short, *visited*), if $\hat{s}$ in the new version of the TV has been reached. Otherwise, s is said to be *unvisited*.

Definition 7: For the route equivalency criterion, the *adaptation rate* of a test case is the percentage of the successfully visited states (Definition 6) during the adaptation of the test case.

Note that the adaptation rate is computed on a per test case basis. To get a perfect adaptation rate, for each state visited by the test case in the older version of the TV, a true equivalent state needs to be visited in the new version of the TV in exactly the same relative order. However, due to the changes in the UIs, additional states may need to be visited in between the consecutive states, which does not affect the adaptation rate.

Definition 8: For the destination equivalency criterion, the *adaptation rate* of a test case is 100% if the destination state (i.e., the final state) is successfully visited (Definition 6) during the adaptation of the test case. Otherwise, the adaptation rate is 0%.

Note that, under the destination equivalency criterion, the average adaptation rate for a collection of test cases is the same as the success rate obtained for the collection. Therefore, when appropriate, we use the two metrics interchangeably in the remainder of the paper.

3) TEST LENGTH OVERHEAD

While the previous two metrics measure the effectiveness of the adaptations, one metric we use to quantify the efficiency is the test length overhead.

Definition 9: The *length of a test case* is the number of RC commands that the test case has, i.e., the length of the path traversed by the test case in the underlying UI model.

Definition 10: The *test length overhead* is the percentage of the additional RC commands used by the adaptation process to automatically adapt a test case with respect to the true minimum number of RC commands required for the same task.

Note that this overhead is computed with respect to the ground truth, rather than the number of RC commands present in the original test case as additional commands may be required for the adaptation.

4) RUNTIME OVERHEAD

We also quantify the efficiency in terms of the runtime overheads. Since the runtime overhead of the proposed approach grows linearly with the number of RC commands present in the test cases, we opted to measure the runtime overhead on a per RC command basis.

We, in particular, define the following itemized runtime overheads, each of which is measured in seconds:

Definition 11: The *screenshot time* is the amount of time required for taking the screenshot of the current screen.

Definition 12: Given the screenshot of a screen, the *selection detection time* is the amount of time required for detecting the selection (Section IV-B), cropping the respective regions out of the screenshot image, and forming the ID-image of the selection (Section IV-C).

Definition 13: Given the ID-image of a selection, the *hashing time* is the amount of time required to compute the hash value for the ID-image (Section IV-C).

Definition 14: Given the hash value for a selection, the *state determination time* is the amount of time required to find the state corresponding to the selection in the underlying UI model (Section IV-C).

Definition 15: Given the ID-image of a selection, the *OCR time* is the amount of time required to recognize the text present in the image, including the text detection as well as the text recognition stages (Section IV-F1).

Definition 16: Given the current state in the underlying model, the *next command determination time* is the amount of time required to determine the next RC command to be executed, including the time required for computing the shortest path (Section IV-E).

Definition 17: Given a state in M, the *equivalent state determination time* is the amount of time required to determine an equivalent state in M' (Section IV-F1).

Definition 18: The *other overheads* include the overheads of the remaining tasks required to glue the partial results together, including the time required to save/load the images.

D. ALTERNATIVE APPROACHES

To carry out comparative studies, we have also implemented two alternative adaptation strategies, namely *random adaptation strategy* and *semi-random adaptation strategy*.

1) RANDOM ADAPTATION STRATEGY

As the name indicates, this strategy mimics the strategies that attempt to adapt the test cases without any prior knowledge of the system under test. For the route equivalency criterion under this strategy, the details of which are given in Algorithm 4, a given test case is executed as it is, on the new version of the TV one RC command after another. After executing an RC command, which moves the older version of the TV from its current state s to the target state s_{target}, the actual state of the UI in the new version is determined (lines 9-10). If the TV turns out to be in an equivalent state of the target state (line 11), then the target state is marked as successfully visited (line 12) and the subsequent RC command in the test case is executed (line 29). Otherwise (line 14), a sequence of random interactions is carried out until an equivalent state of the target state is reached or the maximum number of allowed commands are executed, whichever comes first. If an equivalent state is reached during this search process (line 11), the target state is marked as successfully visited and the subsequent RC command in the test case is executed (line 29). Otherwise, the adaptation process is terminated and marked as a failure (line 23).

Note that the maximum limit on the number of random RC commands that can be executed (i.e., *MAX_ALLOWED* in Algorithm 4) is set on a per target state basis. That is, for each target state, the equivalent state of which cannot be visited by the original RC command, the search process is allowed to execute up to *MAX_ALLOWED* RC commands (in our case, $MAX\_ALLOWED = 150$). We opted to have such a threshold because, otherwise, the experiments would need an undetermined amount of time to finish. Note further that the adaptation rate for the random adaptation strategy under the route equivalency criterion is computed based on the number of states successfully visited before the adaptation process is terminated.

Obtaining destination equivalency with the random adaptation strategy is similar to obtaining route equivalency under the same strategy (Algorithm 4), except for the part where we check whether the equivalent states have been visited or not. More specifically, rather than checking to see whether an equivalent state of the target state has been visited after executing each RC command, we do it only once for the destination state after executing all the RC commands in the test case. If the destination state has not been visited successfully, then a sequence of random RC commands, the maximum count of which grows linearly with the number of original RC commands in the test case (in our case, 10 random RC

Algorithm 4 Random Adaptation (Route Equivalency)

```
1:  function random_adaptation(Test $t$)
2:      Reboot the TV, if needed
3:      $t' \leftarrow []$
4:      while $true$ do
5:          Let $\tilde{s}$ be an equivalent state of the target state $s_{target}$ to be visited in the current iteration
6:          $equivalent\_state\_visited \leftarrow false$
7:          $trial\_cnt \leftarrow 0$
8:          do
9:              $screen \leftarrow take\_screenshot()$
10:             $s' \leftarrow WhereAmI(screen)$
11:             if $s' == \tilde{s}$ then
12:                 Mark $s_{target}$ as successfully visited
13:                 $equivalent\_state\_visited \leftarrow true$
14:             else
15:                 Let $cmd_{rnd}$ be a randomly chosen RC command
16:                 $t' \leftarrow append(t', cmd_{rnd})$
17:                 $execute\_cmd(cmd_{rnd})$
18:                 $trial\_cnt = trial\_cnt + 1$
19:             end if
20:         while (not $equivalent\_state\_visited$) and ($trial\_cnt < MAX\_ALLOWED$)
21:         if not $equivalent\_state\_visited$ then
22:             Terminate the adaptation process
23:             return failure
24:         end if
25:         Let $cmd$ be the next RC command to be executed in $t$
26:         if no $cmd$ left to be executed then
27:             Break out of the outer while loop
28:         end if
29:         $execute\_cmd(cmd)$
30:     end while
31:     return $t'$
32: end function
```

commands per original RC command), are executed. If the destination state is successfully visited during the search, the adaptation process is marked as successful. Otherwise, it is marked as a failure.

2) SEMI-RANDOM ADAPTATION STRATEGY

Semi-random adaptation strategy is designed to mimic the adaptation strategies, which, when a UI element of interest cannot be located in a given screen, can determine an alternative UI element to be interacted with as long as the element is in the current screen. As an example, consider a test case, which is currently visiting a screen that it, indeed, needs to visit. The test case is supposed to click on the `Next` button in the screen, e.g., execute the RC command `OK` on the `Next` button. However, this button has been replaced by a `Proceed` button and placed at a different location on the same screen. As the screen is currently being visited, the UI elements appearing in the screen can be analyzed to determine the most likely alternative to interact with (e.g., the `Proceed` button), which is exactly what is being mimicked by the semi-random adaptation strategy.

Algorithm 5 presents the details of this alternative adaptation strategy for the route equivalency criterion. The algorithm is, indeed, similar to the one given for the random adaptation strategy (Algorithm 4). One difference is that if an equivalent state of the target state s_{target} cannot be visited by executing the original RC command, we check to see if the target state can be reached in the current screen (i.e., if the UI element to be interacted with is in the current screen) (line 14). If so, we leverage the UI model automatically discovered by AdapTV to visit the equivalent state (line 15) by using exactly the same adaptation strategy introduced in Section IV-F3. Note that, for evaluation purposes, this feature is considered to be a part of the semi-random adaptation strategy. Therefore, in these cases, the semi-random adaptation strategy is assumed to have successfully visited the target state and the RC commands executed to this end are counted towards the total number of commands executed by the strategy. If, however, the equivalent state cannot be visited in the current screen, then, as is the case with the random adaptation strategy, a sequence of random RC commands with an upper limit (in our case, $MAX\_ALLOWED = 30$)

Algorithm 5 Semi-Random Adaptation (Route Equivalency)

```
1:  function semi_random_adaptation($Test\ t$, $Model\ M'$)
2:      Reboot the TV, if needed
3:      $t' \leftarrow []$
4:      while $true$ do
5:          Let $\tilde{s}$ be an equivalent state of the target state $s_{target}$ to be visited in the current iteration
6:          $equivalent\_state\_visited \leftarrow false$
7:          $trial\_cnt \leftarrow 0$
8:          do
9:              $screen \leftarrow take\_screenshot()$
10:             $s' \leftarrow WhereAmI(screen)$
11:             if $s' == \tilde{s}$ then
12:                 Mark $s_{target}$ as successfully visited
13:                 $equivalent\_state\_visited \leftarrow true$
14:             else if $\tilde{s}$ can be visited in $screen$ then
15:                 Navigate to visit $\tilde{s}$ by using $M'$
16:                 Populate $t'$ accordingly
17:                 Mark $s_{target}$ as successfully visited
18:                 $equivalent\_state\_visited \leftarrow true$
19:             else
20:                 Let $cmd_{rnd}$ be a randomly chosen RC command
21:                 $t' \leftarrow append(t', cmd_{rnd})$
22:                 $execute\_cmd(cmd_{rnd})$
23:                 $trial\_cnt = trial\_cnt + 1$
24:             end if
25:         while (not $equivalent\_state\_visited$) and ($trial\_cnt < MAX\_ALLOWED$)
26:         if not $equivalent\_state\_visited$ then
27:             Mark $s_{target}$ as unvisited
28:             Mark the adaptation process as failure
29:             Navigate to visit $\tilde{s}$ by using $M'$
30:         end if
31:         Let $cmd$ be the next RC command to be executed in $t$
32:         if no $cmd$ left to be executed then
33:             Break out of the outer while loop
34:         end if
35:         $execute\_cmd(cmd)$
36:     end while
37:     return $t'$
38: end function
```

are executed (lines 20-23). If the equivalent state is visited during this search process (line 11), then the state is marked as successfully visited and the subsequent RC command in the test case is executed (line 31).

Otherwise, the target state s_{target} is marked as unvisited and the adaptation process is considered to have failed (lines 27-28). However, to give a chance to the semi-random strategy to work on the adaptation of the remainder of the test case, rather than terminating the adaptation process, we use the automatically discovered UI model to bring the system from its current state to an equivalent state of s_{target}, such that the adaptation process resumes from where it was left. Note that, since this is something we do solely for evaluation purposes, in these cases, although the adaptation process is considered to have failed, the adaptation rate (Section V-C2) is computed based on the total number of successfully visited states regardless of whether they are visited before or after the adaptation process is marked as a failure.

The destination equivalency criterion under the semi-random adaptation strategy works in a similar manner, except for the part where only the equivalence state for the destination state is checked after executing all the RC commands in a test case. If an equivalent state of the destination state has not been visited, regardless of whether it is in the current screen or not, we execute a sequence of randomly chosen RC commands with a limit on the maximum number of commands that can be executed (in our case, 10 random RC commands per original RC command). If the equivalent state is visited during the search, then the adaptation process is considered to be successful. Otherwise, it is marked as a failure. Note that we don't use the automatically discovered UI model in the destination equivalency criterion to take the TV to the equivalent state of interest, even if it happens to be on the current screen because otherwise the adaptation process (as we have only one equivalent state to visit) would be marked as a failure right away.

E. OPERATIONAL FRAMEWORK

In the experiments, we used exactly the same setup we introduced in Section II. More specifically, the programmable remote controller Tira and the UCD device for taking screenshots were connected to a workstation with a 3.8 GHz Intel(R) Core(TM) i7-10700KF CPU, 32 GB of RAM, and NVIDIA GeForce GTX 1050 Ti graphic card running Windows 11 Education version 21H2. All the screenshots were taken in 4K resolution (3840x2160). The aforementioned workstation was also responsible for executing the proposed adaptation strategy as well as the alternative adaptation strategies, which were all implemented entirely in Python.

Furthermore, we use the following RC commands `[right, left, up, down, OK, menu, back, exit]` both in the model discovery phase and in the adaptation phase. We, in particular, chose to work with these commands because these were the only commands, the behaviors of which depended on the selection. For the rest of the commands, the behaviors were agnostic of the selection. After all, the functionalities of the remaining commands could have been obtained by traversing the user interfaces using the aforementioned list of commands. Moreover, we carried out the model discovery phase with the second stopping criterion of executing a maximum of 20000 RC commands (Section IV-E).

Last but not least, for detecting the selections, we used *OpenCV* [26] v.3.4.1 (Section IV-B). For the older version of the TV, we defined 4 primary selectors and 2 context detectors. And, for the new version, we defined 5 primary selectors and 2 context detectors. For hashing the ID-images, we used *Imagehash* v.4.2.0 (Section IV-C). For text detection, we used *Torch* [27] v.0.4.1 and *Torchvision* [28] v.0.2.2. For text recognition, we used *Torchvision* v.0.2.2 and *NLTK* [29] v.3.5 (Section IV-F1). For equivalent state determination, we used NLTK *v*.3.5 (Section IV-F1).

F. DATA AND ANALYSIS

We first automatically discovered the UI models for both versions of the TV. The model for the older version had 763 states and 4933 edges and the one for the new version had 894 states and 4621 edges. The discovery process for each model took about 39 hours. Note that although the model discovery process can be parallelized by determining the parts of the UI, each of which can be crawled in parallel by using a different setup, such as the one exemplified in Figure 2, we left this as a future work. After all, the UI model for each version of the TV needs to be discovered only once but used for the adaptations of all the test cases.

TABLE 2. Average adaptation rates.

criterion	approach	adaptation rate
route equivalency	random adaptation	0%
	semi-random adaptation	68.64%
	model-based adaptation	100%
destination equivalency	random adaptation	0%
	semi-random adaptation	0%
	model-based adaptation	100%

We first observed that, under the route equivalency as well as the destination equivalency criterion, while the random and semi-random adaptation strategies failed to successfully adapt any of the test cases (i.e., with a success rate of 0%), the proposed approach successfully adapted all the test cases under both criteria (i.e., with a success rate of 100%). Analyzing the adaptation rates (Table 2), we then observed that, for the route equivalency criterion, while the random adaptation strategy was not able to successfully visit any of the states in any of the test cases (with an adaptation rate of 0% for each test case), the semi-random adaptation strategy successfully visited 68.64% of the states, on average. For the destination equivalency criterion, however, both strategies failed to visit any of the destination states (with an adaptation rate of 0% for each test case). The proposed approach, on the other hand, obtained a perfect adaptation rate for each test case regardless of the path equivalency criterion used. That is, all the states for each test case under both the route and destination equivalency criteria were successfully visited during the adaptation processes.

Note that the random strategy was not even able to visit the first states in the test cases. An in-depth analysis revealed that this was because the home screen was changed between the TV versions as presented in Figure 1a and 1b. More specifically, the UI element *Channel Search*, representing the global entry state for the older version of the TV, was moved to a different screen in the new version. And, this required to execute multiple RC commands to be executed (with the shortest sequence being `[up, right, right, OK, right, right, down]`) to get to this UI element starting from the UI element *Input Source*, which represents the global entry state in the new version of the TV (Figure 1b). Note that alternative paths also existed. However, the random adaptation strategy failed to follow any of these paths within the given limits (Section V-D1).

We then studied the adaptation rates of the test cases with respect to their lengths, especially for the semi-random adaptation strategy as the other strategies either successfully adapted all of the test cases (i.e., the model-based strategy) or none of them (i.e., the random strategy). To this end,

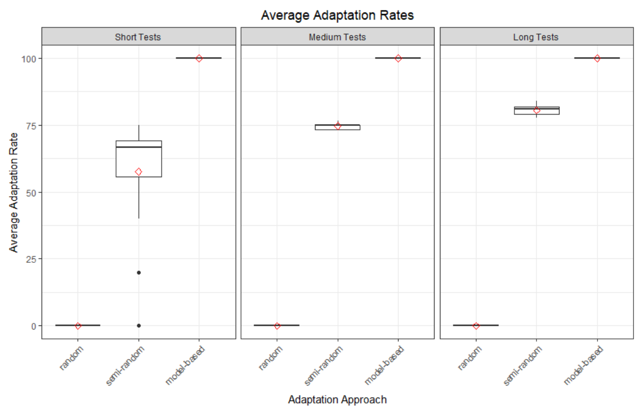


FIGURE 6. Average adaptation rates with respect to the lengths of the test cases.

we grouped the test cases into three categories based on their lengths, namely *short*, *medium*, and *long* test cases. Our clustering criterion was based on having the same or a similar number of test cases in each category as much as possible, which was, indeed, not possible to fully satisfy due to the test cases with the same lengths. Therefore, we opted to label the test cases of length between 1 and 13 as *short*, between 14 and 16 as *medium*, and between 17 and 24 as *long* test cases. All told, we had 54, 33, and 33 test cases in these three categories, respectively.

Figure 6 presents the results we obtained. An interesting observation we make is that as the lengths of the test cases increased, the semi-random adaptation strategy tended to visit more states successfully. More specifically, the average adaptation rates obtained for the short, medium, and long test cases were 57.63%, 74.71%, and 80.58%, respectively. We believe that this was because, as the length of a test case increases, the odds of visiting some UI elements in the screens that have not been changed in between the versions, increase. This observation is also well aligned with our conjecture that the changes in the menu hierarchy typically occur at higher levels of the hierarchy as the lower levels often correspond to the UI elements regarding the fundamental functionalities of the TV, which are typically less likely to get changed.

We next analyzed the test length overheads. Table 3 presents the results we obtained. The columns in this table represent the path equivalency criterion and the test adaptation strategy being used together with the average length of the adapted test cases and the respective test length overhead, on average, respectively. Note that as the random adaptation strategy did not have a chance to work on all of the states visited by the original test case, we opted not to report the results for this strategy as it would be misleading.

The average lengths of the adapted test cases generated by the proposed approach and the semi-random strategy were 48.51 and 155.67, respectively, for the route equivalency criterion and 12.31 and 148.85 for the destination equivalency criterion (Table 3). With these lengths, under the route equivalency criterion, the average test length overheads were 13.39% and 281.06% for the proposed approach and the semi-random strategy, respectively, and, under the destination equivalency criterion, the overheads were 2.12% and 1108 (Table 3).

For the proposed approach, the test length overheads for the destination equivalency criterion were smaller than those for the route equivalency criterion; an average test length overhead of 2.12% vs. 13.39% (Table 3). This was mainly due to the fact that the proposed approach was able to find shorter paths to visit the destination states in the new version of the TV, compared to the paths that need to be followed under the route equivalency criterion.

Analyzing the runtime overheads, which are itemized in Table 4, we observed that the average overheads per RC command were 3.30 and 3.71 seconds in the model discovery and adaptation phases, respectively. Note that all the operations in Table 4, except for the equivalent state determination operation, which is specific to the adaptation phase, are carried out both in the model discovery and adaptation phases. Note further that, on top of these overheads introduced by the proposed approach, there is a runtime cost of 1.77 seconds for executing an RC command using Tira and 2 seconds for waiting in between the consecutive RC commands, which are common between the proposed approach and the original test suite developed by Arcelik.

We first observed that a substantial portion of the overheads was due to taking the screenshots. More specifically, 58.48% and 52.02% of all the overheads in the model discovery and adaptation phases, respectively, were attributed to this task. Note that we solely rely on the UCD device to take the screenshots (Section V-E). Therefore, improvements on these (and similar) devices can profoundly help reduce the runtime overheads of the proposed approach.

We then observed that the next most time-taking operation with an average of 0.66 seconds was to determine the subsequent RC command to execute. This overhead was mostly due to the BFS traversal of the underlying UI model to find the shortest path from the current state to the target state, so that the next command to execute can be determined. Indeed, the computational complexity of this step is $O(|V| + |E|)$ where $|V|$ and $|E|$ are the number of vertices and edges in the UI model, i.e., the running time grows linearly with the size of the UI model.

Last but not least, we observed that OCR took an acceptable amount of time, i.e., 0.13 seconds, on average. This was because we apply OCR only to the cropped images of the selected UI elements, which correspond to a small portion of the entire screen.

VI. THREATS TO VALIDITY

One external threat to study is that we use the TVs produced by a single company, namely Arcelik. However, as Arcelik is one of the largest home appliances manufacturers in Europe

TABLE 3. Average test length overheads.

criterion	approach	average test length	average test length overhead
route equivalency	semi-random adaptation	155.67	281.06%
	model-based adaptation	48.51	13.39%
destination equivalency	semi-random adaptation	148.85	1108.92%
	model-based adaptation	12.31	2.12%

TABLE 4. Amount of time (in seconds) taken by different operations on a per RC command basis.

item	execution time (seconds)
Screenshot time	1.93
Selection detection time	0.24
Hashing time	0.08
State determination time	0.21
OCR time	0.13
Equivalent state determination time	0.41
Next command determination time	0.66
Other overheads	0.05

operating in 100 different countries under 10 different brand names, including Beko and Grundig, we believe that the subject TVs used in the experiments share many common characteristics with the TVs produced by other companies. After all, we have not used any special features provided by the subject TVs in the experiments.

A related threat concerns the representativeness of the test cases used in the experiments. However, half of the test cases were culled from a real test suite, which has been used by Arcelik for testing their latest TVs, encapsulating the decades of experience Arcelik has gained in testing TVs. Furthermore, the UI changes between the versions of the TV used in the experiments were confined to the higher levels of the menu hierarchy and the lower-level menus mostly stayed intact. Based on our experience, this, however, is typical of the changes made in the user interfaces of TVs. As the lower-level menus often correspond to the fundamental functionalities of the TV, they are less likely to get changed between versions. To account for this, thus to avoid any biases in the analysis, we, indeed, selected a subset of the aforementioned test suite by eliminating the redundancies (Section V-B). Furthermore, the other half of the test cases used in the experiments were randomly generated to increase the diversity in the test suite, i.e., to exercise the menus that are not exercised by the real test cases. Indeed, the test cases used in the experiments achieved 100% coverage of the menus; there was at least one test case visiting a UI element in each menu on the TV.

Another potential threat is that we manually defined the ground truths in the experiments, i.e., the sequences of true equivalent states that need to be visited for the adaptations of the test cases. To this end, however, we studied the TV interfaces and carried out small-scale experiments as needed. After all, we validated the ground truths with the respective testing team at Arcelik.

Furthermore, all of the test cases in the experiments were executed on screens with a black background. This was, indeed, the expected behavior of the subject TVs. That is, all the menus were displayed with a black background. This, however, may not be the case for other TVs, i.e., menus can be displayed with complex backgrounds, such as with a background image or with a video (broadcast) playing in the background. On the other hand, the contents visualized in the backgrounds can generally be controlled in testing. Therefore, an appropriate background (such as a solid background) can be chosen for testing the user interfaces. If testing with a more complex background is required, one could filter out the previously known background content from the screenshots. Or, a filtering mechanism based on the opacity of the pixels can be applied as when a TV menu is superimposed on a complex background, the opacity of the background is typically reduced.

Our equivalent state determination approach leverages only the textual information present in the UI elements. If a UI element does not have any text associated with it, then an equivalent state may not be determined. One liable solution is to employ image-based icon classification and use the class labels together with any text associated with these labels in the proposed approach, which we leave as future work. After all, we observe that these cases are rare in the TV domain and image classification is a well-defined and well-studied problem [30].

The screenshot images, especially the ones taken from outside the TV with the use of an external camera, may contain noise. To reduce the noise to the extent possible and to avoid all the issues associated with calibrating the cameras, we used a UCD device in the experiments. However, some noise, which is typically not seen by a naked eye, may still be inevitable. To this end, two hyper-parameters that the proposed approach offers are the size of the hash values to be used with the LSH hashing function and the cutoff value used for determining the equivalency relation between the hash values, both of which can be configured to take the noise into account (Section IV-C).

In our study, we employed Tira and UCD devices to conduct the experiments and evaluate the proposed approach. However, these devices may have some unknown defects. It is, on the other hand, crucial to acknowledge that these two devices are widely used for testing smart TVs at Arcelik. Nonetheless, we have not come across any reported errors by the manufacturers or users of these devices. Furthermore, we conducted a series of preliminary studies to ensure that they are functioning correctly and not affecting the proposed approach.

Furthermore, we only experimented with the English language when carrying out the text detection and extraction operations as well as computing the semantic similarities between texts (Section IV-F1). However, pre-trained models, such as the ones we used for English, also exist for other languages. The performance of these models may, however, vary depending on the language.

VII. RELATED WORK

The literature for consumer electronics, model-based quality assurance, test adaptation, user interface testing, and smart TVs has a long history with many significant contributions. In this section, we provide an overview of the related literature and discuss how our work relates to and/or differs from the existing works.

A. CONSUMER ELECTRONICS

Previous research on consumer electronics has studied various aspects of these devices, including their design, implementation, and testing. For instance, Lin et al. study the user-centered design approaches and the importance of considering the user experience throughout the entire design process [31]. Ommering et al. discuss the implementation of consumer electronics while highlighting the role of hardware and software in the development [32]. Kumar et al. explore different testing approaches for ensuring the quality and reliability of consumer electronics [33]. Other related studies, especially the ones focusing on IoT devices, also exist [34], [35], [36]. We, in particular, contextualize our work within the broader research landscape on consumer electronics.

B. MODEL-BASED QUALITY ASSURANCE APPROACHES

Model-based quality assurance (MQA) approaches have been extensively studied in the literature for various QA-related tasks, including test case generation [37], [38], fault detection [39], and fault localization [40]. Neto et al. survey the MQA approaches for generating test cases in a systematic review work [37]. Garousi et al. empirically study the effectiveness and efficiency of model-based testing in an industrial setup and demonstrate that it leads to improved test coverage and effective fault detection for web and mobile applications [39]. Yilmaz and Williams introduce a model-based fault localization approach to reduce the space of potential root causes for failures, which operates by mimicking the common mistakes made by developers in the form of model mutations [40]. Compared to these works, our work differs in that we present a model-based approach for automatic test case adaptation.

C. TEST ADAPTATION

The topic of test adaptation has been studied mostly in the context of web and mobile applications. Existing test adaptation approaches often utilize the object models, such as the DOM and POM models, to extract information from the screens. Imtiaz et al. propose an approach that uses the differences between the DOM models obtained from two different versions of the same web application to classify the test cases as reusable, re-testable, and obsolete, with the goal of adapting them accordingly [8]. Hammoudi et al. study the reasons for test breakage in the test scripts created for web applications by using record-and-replay tools [6]. Imtiaz et al. survey the literature for test repair approaches [5]. In addition to the DOM- and POM-based approaches, other adaptation approaches have also been studied, which utilize visual rather than structural information [41], [42]. Stocco et al. carry out an empirical analysis comparing DOM-based and visual approaches [42]. Memon and Soffa present a test repair approach for desktop programs, modeling the programs using GUI Control Flow Graphs (G-CFG) and GUI Call Graphs (G-call) [43]. Model-based approaches have, indeed, also been used for test adaptation, but only in the context of web and mobile applications. For example, Imtiaz et al. propose a model-based approach for repairing test cases for evolving web applications [8]. Chen et al. present another model-based approach for automatically identifying and repairing broken test cases for web applications [11].

Our work differs from these works in that, while the existing approaches address the web and mobile platforms, we present a model-based adaptation approach for testing smart TVs. And, from the perspective of test adaptation, smart TV platforms are quite different than the web and mobile platforms. One difference is that, unlike the web and mobile platforms, smart TVs do not necessarily provide object models, such as the POM and DOM models, which report many useful attributes for the UI elements present on the screens, including their types, locations, and labels. Therefore, our approach operates by interpreting the screen images. Another difference is that going from one UI element to another element even on the very same screen on the TV typically requires a sequence of well-planned actions to be taken, e.g., a sequence of arrow buttons on the RC may need to be pressed. To this end, we leverage the automatically discovered UI models in the proposed approach. In the web and mobile platforms, on the other hand, the UI elements can directly be interacted with either by using the references provided by the DOM and POM models or by directly clicking/tapping on the elements.

D. USER INTERFACE TESTING

Banerjee et al. report over 230 publications on UI testing of web applications published between 1991 and 2013 [44], demonstrating that the UI testing of web applications has been extensively studied for nearly three decades, resulting in the development of many sophisticated algorithms, methods, techniques, and tools. Similarly, the UI testing of mobile applications has also been the subject of numerous studies [45], [46], [47], [48]. Although testing web and mobile applications share many common properties, they are also slightly different from each other due to the way the end-users interact with these applications as well as the computing power provided by the underlying platforms. For example, Amalfitano et al. present MobiGUITAR [49] – an approach for UI testing of mobile applications, inspired by GUITAR [50] – an approach for UI testing of web applications. Our work is different in that we present a non-intrusive approach for testing UIs of smart TVs, which poses its own challenges as discussed above.

E. TESTING SMART TVs

Compared to the web and mobile platforms, the testing of smart TVs has so far received less research attention. Ahmed and Bures identify some of the challenges and outline the key components in testing smart TV applications [51]. Cui et al. propose a model-based approach for testing smart TV applications using a Hierarchical State Transition Matrix [52]. Bures et al. develop a model-based UI analysis approach to improve the usability, accessibility, and quality of smart TV applications by detecting possible design flaws in the UIs [53]. Ahmed and Bures present an automated model discovery tool for smart TV applications [54]. However, they use a TV emulator (rather than an actual TV) in the discovery process. Therefore, it is not clear whether some emulator-specific features are used or not. Furthermore, the models are discovered without any particular application (i.e., usage scenario for the models) in mind. We, on the other hand, interact with a physical TV for discovering the UI models, which are specifically designed to be used for test adaptation. Later, Ahmed et al. [55] propose a model-based approach for testing smart TVs that aims to generate effective test cases for fault detection.

The existing works on testing smart TVs, largely focus on general testing problems, rather than specifically addressing the problem of test adaptation. To the best of our knowledge, we are the first to address this problem in the domain of smart TVs.

VIII. CONCLUDING REMARKS AND FUTURE WORK

In this paper, we have introduced a feedback-driven model-based test adaptation approach for testing the user interfaces of smart TVs. From the perspective of the TV software, the proposed approach is a non-intrusive and completely black-box approach, which operates by interpreting the images on the TV screen. Given a test suite, which is known to work on an older version of the TV, we first automatically discover a UI model for both the older version and the new version of the TV, by opportunistically crawling the user interfaces. Then, for each test case in the test suite, we execute it on the older version and determine the path traversed in the respective UI model. Finally, we determine a semantically equivalent path in the UI model discovered for the new version and dynamically execute the selected path in a feedback-driven manner on the new version. The rationale behind having a model-based adaptation approach is to avoid the guesswork as much as possible, which typically occurs when the TV ends up in an unexpected state during the adaptation process. To this end, the proposed approach uses the automatically discovered model to take the TV from its current state to a state of interest.

We have empirically evaluated the proposed approach by using both real test cases, which were developed by Arcelik, and additional test cases, which were automatically generated to improve the diversity. All the experiments were carried out in a setup, which closely mimics the industrial setup used by Arcelik for testing. While the proposed approach successfully adapted all the test cases, the alternative approaches used in the experiments could not adapt any of them.

We believe that this line of research is quite promising. Therefore, we continue to work in this field. One potential avenue for future research is to augment our equivalent state determination approach with image-based icon classification. Another avenue is to apply the proposed approach to other consumer electronics that come with a screen-based user interface, such as washing machines and refrigerators. Yet another promising area is to use the same/similar approaches to migrate the test cases that are written for a specific device to other devices operating in the same domain.

REFERENCES

[1] *Xivt*. Accessed: Oct. 11, 2022-10. [Online]. Available: https://itea3.org/project/xivt.html

[2] M. Mirzaaghaei, "Automatic test suite evolution," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, New York, NY, USA, Sep. 2011, pp. 396–399, doi: 10.1145/2025113.2025172.

[3] A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 1–36, Nov. 2008, doi: 10.1145/1416563.1416564.

[4] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Supporting test suite evolution through test case adaptation," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Apr. 2012, pp. 231–240.

[5] J. Imtiaz, S. Sherin, M. U. Khan, and M. Z. Iqbal, "A systematic literature review of test breakage prevention and repair techniques," *Inf. Softw. Technol.*, vol. 113, pp. 1–19, Sep. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919300990

[6] M. Hammoudi, G. Rothermel, and P. Tonella, "Why do record/replay tests of web applications break?" in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2016, pp. 180–190.

[7] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI crawling-based technique for Android mobile application testing," in *Proc. IEEE 4th Int. Conf. Softw. Test., Verification Validation Workshops*, Mar. 2011, pp. 252–261.

[8] J. Imtiaz, M. Z. Iqbal, and M. U. Khan, "An automated model-based approach to repair test suites of evolving web applications," *J. Syst. Softw.*, vol. 171, Jan. 2021, Art. no. 110841. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302314

[9] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "WATER: Web application test repair," in *Proc. 1st Int. Workshop End-End Test Script Eng.*, New York, NY, USA, Jul. 2011, pp. 24–29, doi: 10.1145/2002931.2002935.

[10] M. Hammoudi, G. Rothermel, and A. Stocco, "WATERFALL: An incremental approach for repairing record-replay tests of web applications," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, New York, NY, USA, Nov. 2016, pp. 751–762, doi: 10.1145/2950290.2950294.

[11] W. Chen, H. Cao, and X. Blanc, "An improving approach for DOM-based web test suite repair," in *Proc. Int. Conf. Web Eng.*, 2021, pp. 372–387.

[12] A. Fırat, M. Y. Azimi, C. Ç. Elgün, F. Erata, and C. Yılmaz, "Model-based test adaptation for smart TVs," in *Proc. 3rd ACM/IEEE Int. Conf. Autom. Softw. Test*, New York, NY, USA, May 2022, pp. 52–53, doi: 10.1145/3524481.3527237.

[13] *TIRA-2.1: Remote Control Receiver/Transmitter*. Accessed: Oct. 11, 2022. [Online]. Available: https://home-electro.com/products/tira-21

[14] *UCD-2 VX1. USB Connected Capture Device*. Accessed: Oct. 11, 2022. [Online]. Available: https://www.unigraf.fi/product/ucd-2-vx1-usb-connected-capture-device

[15] Y. T. Pai, Y. F. Chang, and S. J. Ruan, "Adaptive thresholding algorithm: Efficient computation technique based on intelligent block detection for degraded document images," *Pattern Recognit.*, vol. 43, pp. 3177–3187, Sep. 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320310001342

[16] J. Hosang, R. Benenson, and B. Schiele, "Learning non-maximum suppression," 2017, *arXiv:1705.02950*.

[17] D. Gorisse, M. Cord, and F. Precioso, "Locality-sensitive hashing for Chi2 distance," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 2, pp. 402–409, Feb. 2012.

[18] A. Rivas, P. Chamoso, J. J. Martín-Limorti, S. Rodríguez, F. de la Prieta, and J. Bajo, "Image matching algorithm based on hashes extraction," in *Progress in Artificial Intelligence*, E. Oliveira, J. Gama, Z. Vale, and H. Lopes Cardoso, Eds. Cham, Switzerland: Springer, 2017, pp. 87–94.

[19] H. Wadi, *Step By Step Neural Networks for Image Classification Using Python GUI: A Practical Approach to Understand the Neural Networks Algorithm for Image Classification With Project Based Example*. TURIDA, Feb. 2021.

[20] D. Y. Perwej, S. Hannan, A. Asif, and A. Mane, "An overview and applications of optical character recognition," *Int. J. Advance Res. Sci. Eng.*, vol. 3, pp. 261–274, Jun. 2014.

[21] Y. Baek, B. Lee, D. Han, S. Yun, and H. Lee, "Character region awareness for text detection," 2019, *arXiv:1904.01941*.

[22] *Craft_MLT_25k.pth*. Accessed: Oct. 11, 2022. [Online]. Available: https://drive.google.com/file/d/1Jk4eGD7crsqCCg9C9VjCLkMN3ze8kutZ/view

[23] *Dropbox TPS-ResNet-BiLSTM-Attn-Case-Sensitive.pth*. in *Proc. https://www.dropbox.com/sh/j3xmli4di1zuv3s/AAArdcPgz7UFxIHUuKNOeKv_a?dl=0&preview=TPS-ResNet-BiLSTM-Attn-case-sensitive.pth*

[24] A. Rau, J. Hotzkow, and A. Zeller, "Transferring tests across web applications," in *Proc. Int. Conf. Web Eng.*, 2018, pp. 50–64.

[25] Kaggle. *Googlenews-Vectors-Negative300*. Accessed: Dec. 13, 2022. [Online]. Available: https://www.kaggle.com/datasets/leadbest/googlenewsvectorsnegative300

[26] G. Bradski, "The openCV library," *Dr. Dobb's J., Softw. Tools Prof. Programmer*, vol. 25, no. 11, pp. 120–123, 2000.

[27] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A MATLAB-like environment for machine learning," in *Proc. BigLearn, NIPS Workshop*, 2011.

[28] S. Marcel and Y. Rodriguez, "Torchvision the machine-vision package of torch," in *Proc. 18th ACM Int. Conf. Multimedia*, New York, NY, USA, Oct. 2010, pp. 1485–1488, doi: 10.1145/1873951.1874254.

[29] S. Bird, E. Klein, and E. Loper, *Natural Language Processing With Python: Analyzing Text With the Natural Language Toolkit*. Newton, MA, USA: O'Reilly Media, Inc., 2009.

[30] R. Ponnusamy, S. Sathyamoorthy, and K. Manikanda, "A review of image classification approaches and techniques," *Int. J. Recent Trends Eng. Res.*, vol. 3, no. 3, pp. 1–5, 2020.

[31] K.-Y. Lin, A. Yu, P.-C. Chu, and C.-F. Chien, "User-experience-based design of experiments for new product development of consumer electronics and an empirical study," *J. Ind. Prod. Eng.*, vol. 34, pp. 1–16, Sep. 2017.

[32] R. V. Ommering, F. V. D. Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, Mar. 2000.

[33] K. Kumar, S. Kumar, and L. K. Singh, "Evaluating technologies for reliable software in consumer electronics: A survey of component failure modeling," *IEEE Consum. Electron. Mag.*, vol. 8, no. 6, pp. 56–61, Nov. 2019.

[34] A. Albahli and A. Andrews, "Model-based testing of smart home systems using EFSM and CEFSM," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2021, pp. 1824–1829.

[35] T. Alladi, V. Chamola, B. Sikdar, and K. R. Choo, "Consumer IoT: Security vulnerability case studies and solutions," *IEEE Consum. Electron. Mag.*, vol. 9, no. 2, pp. 17–25, Mar. 2020.

[36] M. Park, H. Jang, T. Byun, and Y. Choi, "Property-based testing for LG home appliances using accelerated software-in-the-loop simulation," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng., Softw. Eng. Pract.*, Jun. 2020, pp. 120–129.

[37] A. Neto, R. Subramanyan, M. Vieira, and G. Travassos, "A survey on model-based testing approaches: A systematic review," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2007, pp. 31–36.

[38] E. Erdem, K. Inoue, J. Oetsch, J. Pührer, H. Tompits, and C. Yilmaz, "Answer-set programming as a new approach to event-sequence testing," in *Proc. 3rd Int. Conf. Adv. Syst. Test. Validation Lifecycle*, 2011, pp. 1–10.

[39] V. Garousi, A. B. Keleş, Y. Balaman, Z. Ö. Güler, and A. Arcuri, "Model-based testing in practice: An experience report from the web applications domain," *J. Syst. Softw.*, vol. 180, Oct. 2021, Art. no. 111032. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001291

[40] C. Yilmaz and C. Williams, "An automated model-based debugging approach," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, New York, NY, USA, Nov. 2007, pp. 174–183, doi: 10.1145/1321631.1321659.

[41] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "PESTO: Automated migration of DOM-based web tests towards the visual approach," *Softw. Test., Verification Rel.*, vol. 28, no. 4, p. e1665, Jun. 2018.

[42] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA, Oct. 2018, pp. 503–514, doi: 10.1145/3236024.3236063.

[43] A. M. Memon and M. L. Soffa, "Regression testing of GUIs," in *Proc. 9th Eur. Softw. Eng. Conf. 11th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, New York, NY, USA, Sep. 2003, pp. 118–127, doi: 10.1145/940071.940088.

[44] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1679–1694, Oct. 2013.

[45] S. Zein, N. Salleh, and J. Grundy, "A systematic mapping study of mobile application testing techniques," *J. Syst. Softw.*, vol. 117, pp. 334–356, Jul. 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216300140

[46] R. Coppola, E. Raffero, and M. Torchiano, "Automated mobile UI test fragility: An exploratory assessment study on Android," in *Proc. 2nd Int. Workshop User Interface Test Autom.*, New York, NY, USA, Jul. 2016, pp. 11–20, doi: 10.1145/2945404.2945406.

[47] S. Talebipour, Y. Zhao, L. Dojcilovic, C. Li, and N. Medvidovic, "UI test migration across mobile platforms," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 756–767.

[48] D. Amalfitano, A. R. Fasolino, P. Tramontana, and B. Robbins, *Testing Android Mobile Applications: Challenges, Strategies, and Approaches* (Advances in Computers), vol. 89, A. Memon, Ed. Amsterdam, The Netherlands: Elsevier, 2013, pp. 1–52. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124080942000011

[49] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, Sep. 2015.

[50] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An innovative tool for automated testing of GUI-driven software," *Automated Softw. Eng.*, vol. 21, no. 1, pp. 65–105, Mar. 2014.

[51] B. Ahmed and M. Bures, "Testing of smart TV applications: Key ingredients, challenges and proposed solutions," in *Proc. Future Technol. Conf.*, 2018, pp. 241–256.

[52] K. Cui, K. Zhou, H. Song, and M. Li, "Automated software testing based on hierarchical state transition matrix for smart TV," *IEEE Access*, vol. 5, pp. 6492–6501, 2017.

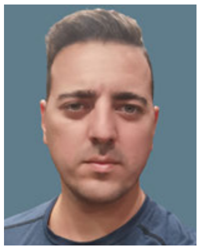
[53] M. Bures, M. Macik, B. S. Ahmed, V. Rechtberger, and P. Slavik, "Testing the usability and accessibility of smart TV applications using an automated model-based approach," *IEEE Trans. Consum. Electron.*, vol. 66, no. 2, pp. 134–143, May 2020.

[54] B. S. Ahmed and M. Bures, "EvoCreeper: Automated black-box model generation for smart TV applications," *IEEE Trans. Consum. Electron.*, vol. 65, no. 2, pp. 160–169, May 2019, doi: 10.1109/TCE.2019.2907017.

[55] B. S. Ahmed, A. Gargantini, and M. Bures, "An automated testing framework for smart TV apps based on model separation," 2020, *arXiv:2002.00404*.

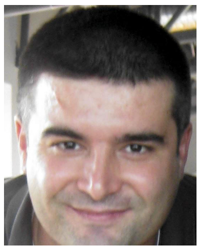
MOHAMMAD YUSAF AZIMI received the B.S. degree in computer engineering from Sakarya University, in 2018. He is currently pursuing the Ph.D. degree with the Computer Science Department, Sabanci University. He has extensive experience in software engineering. His research interest includes automated test adaptation for consumer electronics with graphical user interfaces. He participated in multiple projects in his research field.

CELAL CAGIN ELGUN received the B.Sc. degree from Dokuz Eylul University. He has 15 years of experience in Arçelik, where he is currently the Head of Test Development/Chapter Lead with the Central Research and Development. He is leading the team to develop in-house digitalization and automation infrastructure solutions for the test phase of Arçelik's product designs and development process.



ATIL FIRAT received the degree from the Sefakoy Anatolian High School, Bilkent University. He is currently a Senior Lead Engineer with the Test Team, Arçelik, where he is responsible for test development and automation of tests. Arçelik's employees take part in the software of the systems that meet employees' needs—avoiding outsourcing contributes to the company's development of technologies by using in-house solutions. Before joining Arçelik, he was a .Net Developer with Vera Cash Registers and a Cash Register Software Engineer. Before joining Vera Cash Register, he was a Software Engineer with Defense Technologies.

FERHAT ERATA (Member, IEEE) is currently pursuing the Ph.D. degree in computer science with Yale University. He served as a software research engineer for various research and development projects consortia in the aviation and automotive industry. He is an Applied Scientist Intern with the Automated Reasoning Group, Amazon Web Services, developing efficient methods for checking the concurrency of distributed storage systems. His research interests include the automatic detection and repair of side-channel vulnerabilities in cryptographic code and reasoning about bit-vector logic.



CEMAL YILMAZ (Member, IEEE) received the B.S. and M.S. degrees in computer engineering and information science from Bilkent University, Ankara, Turkey, in 1997 and 1999, respectively, and the Ph.D. degree in computer science from the University of Maryland, College Park, in 2005. From 2005 to 2008, he was a Postdoctoral Researcher with the IBM Thomas J. Watson Research Center, Hawthorne, NY, USA. He is currently an Associate Professor of computer science with the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey. His current research interests include software quality assurance and software/system security.